BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING
FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS
ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

DESIGN OF AUTONOMOUS VEHICLE SIMULATOR
NÁVRH SIMULÁTORU AUTONOMNÍHO DOPRAVNÍHO PROSTŘEDKU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Petr Machač**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. Stanislav Věchet, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF MECHANICAL ENGINEERING

Specification Master's Thesis

Department:	Institute of Solid Mechanics, Mechatronics and Biomechanics
Student:	**Bc. Petr Machač**
Study programme:	Applied Sciences in Engineering
Study branch:	Mechatronics
Supervisor:	**doc. Ing. Stanislav Věchet, Ph.D.**
Academic year:	2019/20

Pursuant to Act no. 111/1998 concerning universities and the BUT study and examination rules, you have been assigned the following topic by the institute director Master's Thesis:

Design of autonomous vehicle simulator

Concise characteristic of the task:

The aim of this thesis is to develop a simulation environment of autonomous vehicle using the dynamic engine Box2D. The scope for the simulator is to cover basic interactions between the vehicle and surrounding obstacles which are represented by both the static (buildings) and dynamic (walking pedestrians) obstacles. One of the sub goals is to develop an API for controling the simulation environment which will be further used for AI methods development and testing.

Goals Master's Thesis:

1. Perform a search of available simulation tools for autonomous vehicles.
2. Design a simulator architecture to fit a dynamic Box2D simulation environment.
3. Implement necessary API interface to control simulator environment.
4. Test the desired functionality.

Recommended bibliography:

PARBERRY, I., Introduction to Game Physics with Box2D. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2013.

Deadline for submission Master's Thesis is given by the Schedule of the Academic year 2019/20

In Brno,

L. S.

prof. Ing. Jindřich Petruška, CSc. Director of the Institute	doc. Ing. Jaroslav Katolický, Ph.D. FME dean

ABSTRAKT

Tato práce se zabývá simulačními prostředky pro vývoj algoritmů pro řízení autonomních automobilů. V zásadě lze rozdělit na dvě části, na rešeršní, teoretickou, a praktickou, vývojovou. V té prvně zmíněné je uveden přehled dostupných nástrojů pro simulaci autonomních vozidel, jedná se jak o nástroje open-sourcové tak placené. Dále se v teoretické části popisuje princip a nástroje, resp. enginy pro řešení dynamických rovnic na počítači. Důraz je kladen na fyzikální engine Box2D který je dle zadání této práce využit ve druhé části teze pro vývoj vlastního prostředí simulujícího autonomní automobil.

SUMMARY

This thesis deals with simulation tools for algorithm development for autonomous automobiles control. The thesis can be divided into two parts, first a research of currently available simulation tools, both open-source and proprietary is made. Then ways of modelling of physical systems and dynamic equations solving engines are described. An emphasis is given on a Box2D physical engine which is then used in the second part of the thesis as a base for own environment for autonomous vehicle simulation.

KLÍČOVÁ SLOVA

Autonomní automobil, Box2D, Simulátor, Q-učení, Počítačová fyzika

KEYWORDS

Autonomous Vehicle, Box2D, Simulator, Q-Learning, Computer physics

BIBLIOGRAPHIC CITATION

MACHAČ, Petr. *Návrh simulátoru autonomního dopravního prostředku* [online]. Brno, 2020 [cit. 2020-05-04]. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/124887. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav mechaniky těles, mechatroniky a biomechaniky. Vedoucí práce Stanislav Věchet.

ROZŠÍŘENÝ ABSTRAKT

ÚVOD

Autonomní automobil je takové motorové vozidlo, které pro svou jízdu nevyžaduje zásah řidiče. Vytvořit plně autonomní vůz je extrémně náročný úkol, který se v době vzniku této práce nepodařilo uspokojivě splnit. V praxi se ale využívá šestistupňový ukazatel autonomie vozidla, automobily jsou rozděleny do úrovní 0 – 5 kde 0 je auto bez jakýchkoliv asistenčních systémů a 5 je plně autonomní vůz, které zatím existují jen v malých počtech experimentálních vozidel.

Vývoj autonomního automobilu vyžaduje značné množství finančních prostředků, výpočetní kapacity apod. Automobily musejí splňovat celou řadu bezpečnostních a jiných parametrů. Z těchto důvodů není možné při vývoji přeskočit fázi modelování a simulací a rovnou přejít na testování v reálném prostředí.

Simulační prostředky, přestože mohou být velice drahé stále představují jen zlomek nákladů na vytvoření reálného automobilu. Nehledě na zvýšenou bezpečnost, jsou již známy případy kdy experimentální autonomní automobil kvůli chybě algoritmu usmrtil účastníky silničního provozu. Toto se při simulacích ve virtuálním prostředí nemůže stát.

Tato diplomová práce provádí rešerši aktuálně dostupných simulačních nástrojů, následně popisuje základy počítačové fyziky, kterou je nutno při simulaci brát v potaz (pro zjednodušení zejména 2D fyziky) a následně popisuje postup při vývoji vlastního simulačního prostředí vhodného pro aplikaci různých algoritmů umělé inteligence, především zpětně posilované učení, konkrétně Q-učení. Prostředí je vyvinuto s pomocí fyzikálního 2D enginu Box2D

POPIS ŘEŠENÍ

Vyvinutý simulátor obsahuje tři různé typy objektů či tři různé entity, určitý počet statických předmětů reprezentující budovy rozložené do jednoduché sítě, určitý počet dynamických předmětů reprezentující chodce, kteří se mohou pohybovat v blízkosti budov a na určitých místech přecházet od jedné budovy ke druhé, podobně jako reální chodci. A jednoho automobilu. Automobil má rozměry podle vozu Tesla Model 3. Jedná se o dynamický objekt, který slouží jakožto hlavní agent pro vývoj algoritmů umělé inteligence. Fyzikální engine Box2D je nastaven tak aby hlídal kolize mezi automobilem a ostatními objekty a následně jsou z informací o těchto kolizích vyvozeny důsledky. Jelikož při vývoji simulátoru byl kladen důraz na aplikaci tzv. *Reinforcement learning* (možný český ekvivalent je: *Zpětně posilované učení*), konkrétně *Q-učení*, jehož základem je učení s pomocí velkého množství (řádově tisíců i desetitisíců) tzv. *epizod* je především nutné zajistit, aby na konci každé epizody bylo prostředí resetováno do nějakého počátečního stavu. Právě k tomuto slouží informace o kolizích, na základě těchto zjištění dochází k onomu resetu celého prostředí a začátku nové epizody.

SHRNUTÍ A ZHODNOCENÍ VÝSLEDKŮ

V rámci této diplomové práce bylo vytvořeno prostředí pro simulaci autonomního automobilu, resp. pro aplikaci algoritmů umělé inteligence, zvláště v oblasti tzv. *reinforcement learning*. V případě tohoto simulátoru se snaží vyhnout chodcům, vydržet bez kolize po daný počet kroků nebo vyjet s agentem (automobilem) mimo vyznačenou oblast (záleží na nastavení, co je považováno za úspěšné splnění úkolu).

V rámci testování funkčnosti simulátoru byly navrženy dvě možné reprezentace stavového prostoru. Účelem testování nebylo naučit automobil (agenta) se řádně pohybovat a učit ve svém prostředí, ale zjistit, zda je navržený simulátor pro tento úkol vhodný. Více se osvědčila reprezentace, spočívající v tom, že přední zorné pole vozidla bylo rozděleno na n výsečí po $180/n$ stupních, v každé výseči se našel chodec (pokud zde nějaký byl) a byl ohodnocen podle vzdálenosti stavy 0, 1, 2 podle vzdálenosti od vozidla (0 – daleko, výchozí stav 1 – ve „střední“ vzdálenosti 2 – blízko). U této reprezentace bylo zjištěno, že se vytváří korektně, správně se volí stavový vektor a na jeho základě se správně vybírá akce z Q-tabulky.

Na základě těchto zjištění je možné konstatovat, že navržený simulátor je vhodný pro budoucí aplikaci metod umělé inteligence.

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This master's thesis is my own work and contains nothing which is the outcome of work done in collaboration with others.

Petr Machač

Brno . . . .

I would like to thank my supervisor, doc. Ing. Stanislav Věchet, Ph. D. for his guidance, help and support during the development of this thesis. Also, I would like to express my gratitude to my colleagues in our mechatronics study group for help and support. And last, but not least, I want to thank to my family for their endless support during my whole studies

Petr Machač

CONTENTS

INTRODUCTION

An autonomous vehicle is an automobile that uses for its movement on roads only data obtained from a variety of mechatronics sensors such as GPS locators, radars, lidars, or cameras and process them using artificial intelligence so it is completely independent of the driver's interventions. This car is called a fully autonomous vehicle or driverless car and such vehicles do not yet exist but there are available cars with some degrees of autonomy. The levels of autonomy were defined by the Society of Automobile Engineers (SAE) in 2014. The vehicle is labelled by a number from 0 to 5, where level 0 means no automation, and level 5 means full automation, and each level is described below [1].

To reduce costs and time needed for car development and avoid risks related to incorrectly designed control algorithms the car manufacturers are trying to simulate as much of the car as possible before physical production. It is possible to model forces acting on the car using finite element method (FEM) or dynamic properties of a car as a multi-body system in programs such as MSC Adams or simulation of various control algorithms in MATLAB/Simulink

Modern simulation and analytical tools allow us, in theory, to completely skip the beta-testing phase (tests of masked cars in normal traffic. It is in this phase where spy photos of future models often appear). However, there is only one publicly known occasion when this phase was skipped – during the development of the Tesla Model 3 [2].

The simulation of autonomous vehicles is mainly about whether the above-mentioned sensors communicate well with each other and with control units and whether artificial intelligence is properly set up. The sensors can be either simulated or connected to the control unit in Hardware-In-The-Loop (HIL) testing. Data obtained from these sensors enter the processing and the output can be for example the value of kinematic quantities position, velocity, and acceleration, or planned trajectory.

The aim of this diploma thesis is to design a 2D environment that is supposed to be able to connect to an artificial intelligence engine which would be used to develop algorithms for autonomous driving.

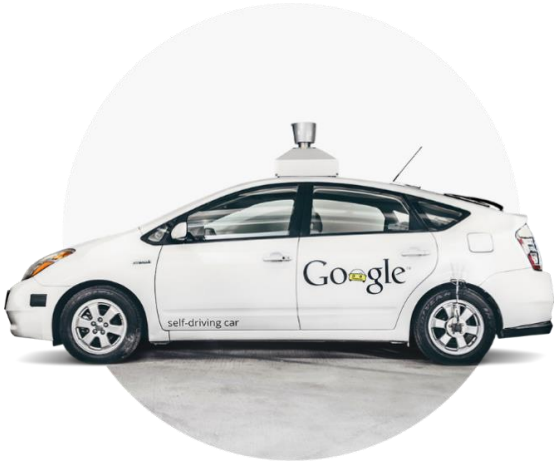

Fig. 1: Self-Driving Toyota Prius used in Google Waymo Fleet [3]

Companies engaged in the development of autonomous vehicles

Arguably the most known organization doing a research in this area is Alphabet (Google) subsidiary Waymo. The company has a fleet of driverless cars operating on stage 4 autonomy.
But basically, all major car manufacturers such as VW or Toyota are researching autonomous vehicles, Ford Motor Company announced that they would like to get into market a stage 4 car in 2021 [4]. Another example can be Tesla Motors, which has officially reached stage 2 autonomy with its Tesla Autopilot functionality.

But not only car manufacturers are considering driverless cars, for example, alternative-taxi-platforms such as Uber or Lyft (Lyft Level 5 division) have their programs concerned on autonomous vehicle development [5][6]. What concerns Uber, there is an unfortunate primacy associated with the company. On March 18th, 2018, its experimental self-driving Volvo XC90 killed a pedestrian pushing a bicycle in Tempe, Arizona. This was the first fatality involving such a vehicle [7].

Levels of autonomy definition

As mentioned above there are six levels of autonomy from 0 to 5 (0 – no autonomy, 5 – full autonomy). In this section a short description of each level is written:

- Level 0 - no autonomy
  Driving is fully dependent on a driver; the car produces no signals or at most. they have an informative or warning character such as beeping sounds during driving at a reverse
- Level 1 – Driver-assist systems (“Hands-on”)
  Cars with this autonomy level use some kind of information about its environments form sensors, assist systems like lane assist or adaptive cruise control are considered as a level 1 autonomy systems
- Level 2 – Partial autonomy (“Hands off”)
  The car is able to - under given circumstances - drive and control the speed of the vehicle (level 1 cars can do either of that but not both). An instance of these systems is, for example, advanced adaptive cruise control or park assist for longitudinal parking
- Level 3 – Conditional autonomy (“Eyes off”)
  The vehicle is capable of taking control over driving in basic situations, the driver does not have to be fully concentrated on driving.
  Car manufacturers usually try to “skip” this level and develop level 4 cars
- Level 4 – High autonomy (“mind off”)
  The main difference between level 3 and level 4 is that the car must be able to safely stop or allow the driver to take full control over the vehicle. But similar to level 3 the vehicle is able to drive automatically during “more complex” situations. There can be zones given by local legislature where it is allowed to drive on autopilot, but there are still situations where the driver is irreplaceable, for example, complex highway crossroads.
  Google Waymo fleet operates on this level.
- Level 5 – Full autonomy
  As the name suggests the vehicle is capable of solving all situation which can occur during driving automatically. Driver’s intervention is not supposed to be necessary

and usually it is considered that it will be impossible, due to the fact that the car is not supposed to have a steering wheel.
It is also suggested that the fully autonomous car can be in general safer than non-autonomous vehicles because a clear majority of car accidents are caused by a human error.
The cars are developed in close cooperation with software companies such as nVidia because these cars need a very high level of artificial intelligence.

1 EXISTING SIMULATORS

Every company engaged in the development of autonomous vehicles uses various simulation environments. Some of them developed their proprietary software suitable exactly for their needs, or they use any third-party-developed tools (usually they use both approaches). This part of the thesis covers the most known simulators which were designed specifically to research autonomous driving. In theory, even racing or adventure games can be used to simulate autonomous cars.

Some of these simulators are open-sourced (for example CARLA or AirSim, which are under MIT license which allows use for commercial purposes and can be used even for proprietary – or closed-code – software) [8].

1.1 AirSim

Microsoft AirSim (Aerial Informatics and Robotics Simulation) is an open-sourced cross-platform system developed by Microsoft Corporation released in 2017. It was introduced in paper *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles* (Authors Shah et al.) [9]. Its main application is for simulation of drones but it can be used for autonomous vehicle simulation as well.

It is not a standalone software, it is a plugin that is imported into a given game engine. It is capable of creating autonomous models using methods of machine learning, and artificial intelligence (computer vision, deep learning, imitation learning and reinforcement learning), it also supports Hardware-in-the-loop (HIL) systems.

The system can be downloaded from GitHub and it is written in C++ but APIs used to retrieve information are accessible using Python, C#, and Java and it is available for operating systems Windows 10 and Linux. Originally it is created as an Unreal engine plug-in, but experimental Unity release is available as well.

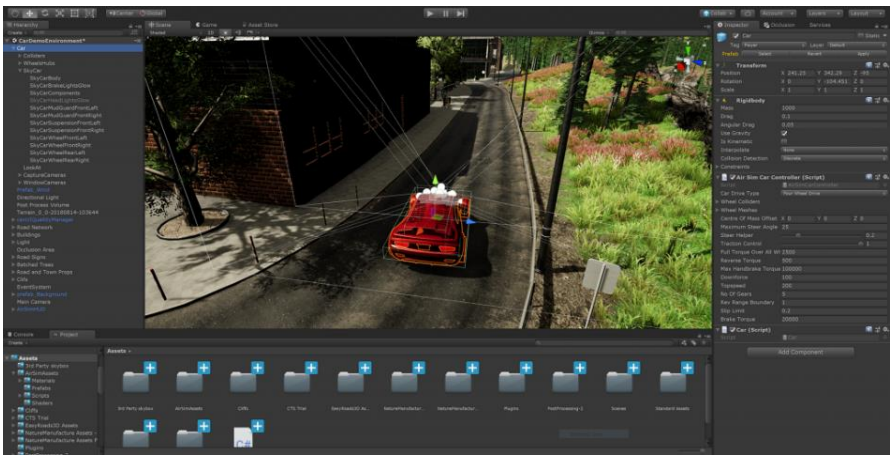

Fig. 2: Microsoft AirSim ran in the UNITY Engine [10]

1.2 Udacity Self-Driving Car Simulator

This simulator was developed by Udacity, Inc. a for-profit educational organization for purposes of Udacity's "Self-Driving Car Nanodegree" which is an educational project

aimed to teach the general public how to program autonomous vehicles using deep learning. The simulator is available on GitHub and it is based on the Unity engine.
The Self-Driving Car Nanodegree is teaching how to use a camera for finding lanes on the road, convolutional neural network to classify traffic signs, and also using a neural network to drive a vehicle and also sensor fusion using extended Kalman filters.
Furthermore, the Nanodegree is supposed to teach how to precisely determine vehicle localization, also how to plan a trajectory for a car and how to actuate the vehicle (accelerating, braking, steering…) using PID controller and the ultimate project goal is to teach basics of system integration [11].

1.3 VIRES Virtual Test Drive

VIRES Virtual Test Drive (or VTD for short) is a paid software developed by American company MSC Software Corporation. In general, it is similar to IPG CarMaker.

It consists of several modules, for example, Road Designer, which user interface is similar to that of MathWorks RoadRunner. The goal of this tool is to create a scene with a road and possibly some other entities such as trees, buildings, barriers, etc. on which the driving scenarios can be tested, but the user can also use some pre-prepared city layouts. It supports both the import and export of OpenDRIVE *.xodr* files.

When the scene is ready the user can move to Scenario Designer in which very complex scenarios can be set up. Settings of the scenario, such as visibility, time of the day, weather condition, or set of sensors mounted on a car can be modified during the simulation runtime.

What concerns control of vehicles it is possible to either use a built-in VTD driver or import control algorithm from third-party tools such as MATLAB/Simulink or ADTF or to merge both third-party and built-in control toolchains.

There is no Windows release of Virtual Test Drive, it is possible to run only using Linux distributions.

As mentioned above VTD is modular and scalable software, which means that the price of a license is hard to tell and there is no publicly known price-list. The offer with the price is available only on request [12].

Fig. 3: Example of a Virtual Test Drive environment [12]

1.4 CarMaker, TruckMaker, and MotorcycleMaker

These applications are developed by German company IPG Automotive. All of them are focused on virtual test driving, as the names suggest CarMaker is aimed at passenger cars and light-duty vehicles, MotorcycleMaker on two-wheeled vehicles, and TruckMaker on heavy-duty vehicles. Using these software tools, it is possible to generate 3D environments apart from roads and cars including the entire surroundings such as buildings, traffic signs, traffic lights, plants, trees, etc. These programs can be used within the whole car software and hardware development process including Vehicle-in-the-Loop (often abbreviated to ViL) [13].

Fig. 4: CarMaker environment [13]

1.5 MATLAB Automated Driving Toolbox

Automated Driving Toolbox (or just ADT for short) is a tool developed by American company MathWorks, Inc. as a module for its tool MATLAB. It allows user to design and simulate Advanced Driver Assist Systems (ADAS). Using this toolbox, it is possible to generate route layout define the number of lanes on a route, place obstacles, fill the environment with agents (or *Actors* using MATLAB terminology). Actor movement is realized either by user given waypoints (this is used mainly to simulate more realistic traffic flow) or to develop custom path planning algorithms using artificial intelligence, dynamic programming, model predictive control and so on (this approach is usually used for moving user-controlled vehicle – in ADT called Ego). It is also possible to mount the vehicle with sensors and cameras and achieve data from these sensors and analyse them [14].

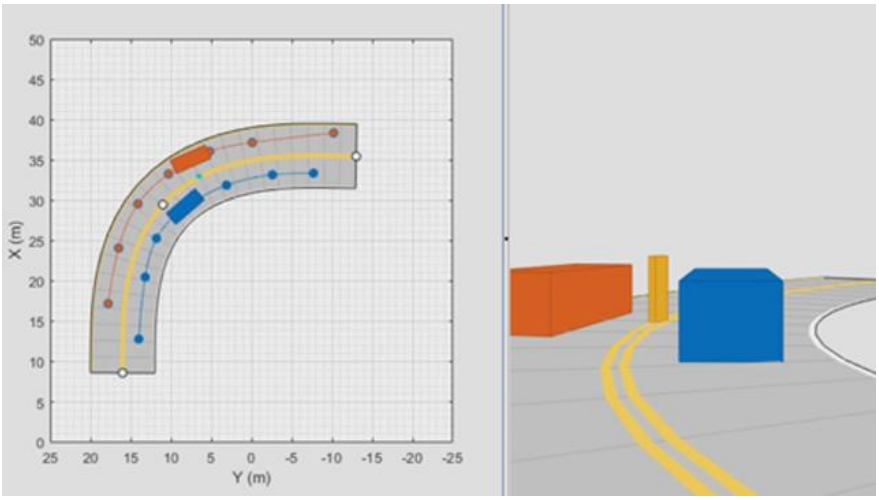

Fig. 5: Road Scenario Designer (RSD) in MATLAB Automated Driving Toolbox [15]

In MATLAB release R2020a a new possibility of visualization using the UNREAL engine was introduced. It is possible to generate a 3D scene from Road Scenario Designer or to use pre-defined scenes representing e.g. a parking lot [17].

1.6 MathWorks RoadRunner

MathWorks RoadRunner (from 2017 to 2020 known as VectorZero Roadrunner) is a 3D designer of road environments developed by American Company VectorZero, founded in 2017. It can be used for autonomous vehicle simulation. An expansion pack called PropKit is available which contains traffic lights, signs, barriers and it is planned to place cars and pedestrians into the world.

User can either create his urban layout or one can import geographic information system (or GIS for short) data and place the roads accordingly to the real environment. Also, the usage of existing LiDAR data is possible to allow the user to place roads and props as realistically as possible.

Created layouts can then be exported to either the Unity or the UNREAL game engine or to industry simulation tool OpenDRIVE so the further simulation can be performed on these roads.

In early 2020 the VectorZero became a part of MathWorks Company and the plan is to make RoadRunner compatible with MATLAB and Simulink in near future [16]

Even though there is a 30-days free demo version available, huge disadvantage of this software is its high purchase cost, before purchasing by MathWorks one license was sold for 12,500 US dollars (about 300,000 Czech crowns) after the acquisition the price is available on-demand but it is reasonable to expect the price to be similar.

Fig. 6: Complex roundabout designed in VectorZero RoadRunner environment [16]

1.7 OPAL-RT

OPAL-RT Technologies, Inc. is a Canadian company founded in 1997. Its main focus is on real-time simulations for automotive, aerospace, power electronics, and power generation.

They developed a platform called RT-LAB Orchestra with ADAS service (Advanced Driver Assistance System Application) which are aiming to lightning, cruise control, braking, and traffic warnings automation.

RT-LAB orchestra is performing its simulations on three platforms, simulators based on Intel Xeon E5 processors, High-Performance Computing (HPC), and Cloud computing.

OPAL-RT is a commercial company and therefore their solutions are not open-source [19].

1.8 CARLA

CARLA (abbreviation from Car Learning to Act) is an open-source simulator for autonomous driving research. It was developed in Computer Vision Center, which is a non-profit research centre by the Generalitat de Catalunya and the Universitat autonòma de Barcelona, Spain in cooperation with Intel Labs and Toyota Research Institute.

It was introduced in paper *CARLA: An Open Urban Driving Simulator* published in 2017 (Authors: Alexey Dosovitskiy et al.) [21].

This simulator is based on the game engine UNREAL Engine 4 (same as Microsoft AirSim, but unlike the AirSim it is not only UNREAL plugin, but standalone application) which is available for Microsoft Windows and GNU/Linux and is free for non-commercial use.

CARLA is used to study several approaches of autonomous driving such as a classic modular pipeline or deep network via imitation learning.

CARLA simulator is designed as a server-client system. The server (called also the simulator) is responsible for solving the vehicle movement physics and rendering all vehicles, agents, sensors or environments. The client (the Python API) sends commands and meta-commands to the server and receives sensor data. The difference between commands and meta-commands is that commands control the vehicle (its steering,

accelerating and braking) and meta-commands control the behaviour of the server or is changing the environmental properties such as weather condition (user can set different levels of rain, wind, or illumination) or it can modify the density of cars and pedestrians.

The simulator requires also two available Transmission Control Protocol (TCP) ports, by default 2000 and 2001. The first one controls simulation using rpclib (Remote Procedure Call - RCP – a protocol that takes care of both a client and a server implementation), the latter provides streaming of the sensor data. This must be achieved via firewall settings.

To run CARLA, it is necessary to either download it from GitHub (various releases are available for both Microsoft Windows and Linux) as a ready-to-run application, the size of the repository is about 2,5 GB. The simulator then can be launched by running executable file CarlaUE4.exe (for Windows) or using command ./CarlUE4.sh (for Linux) or download its source code and built it from source.

After that, a window with a view over a city is launched. User can move around the world using WASD keys but it is impossible to interact with the world. For that the user has to use the Python API.

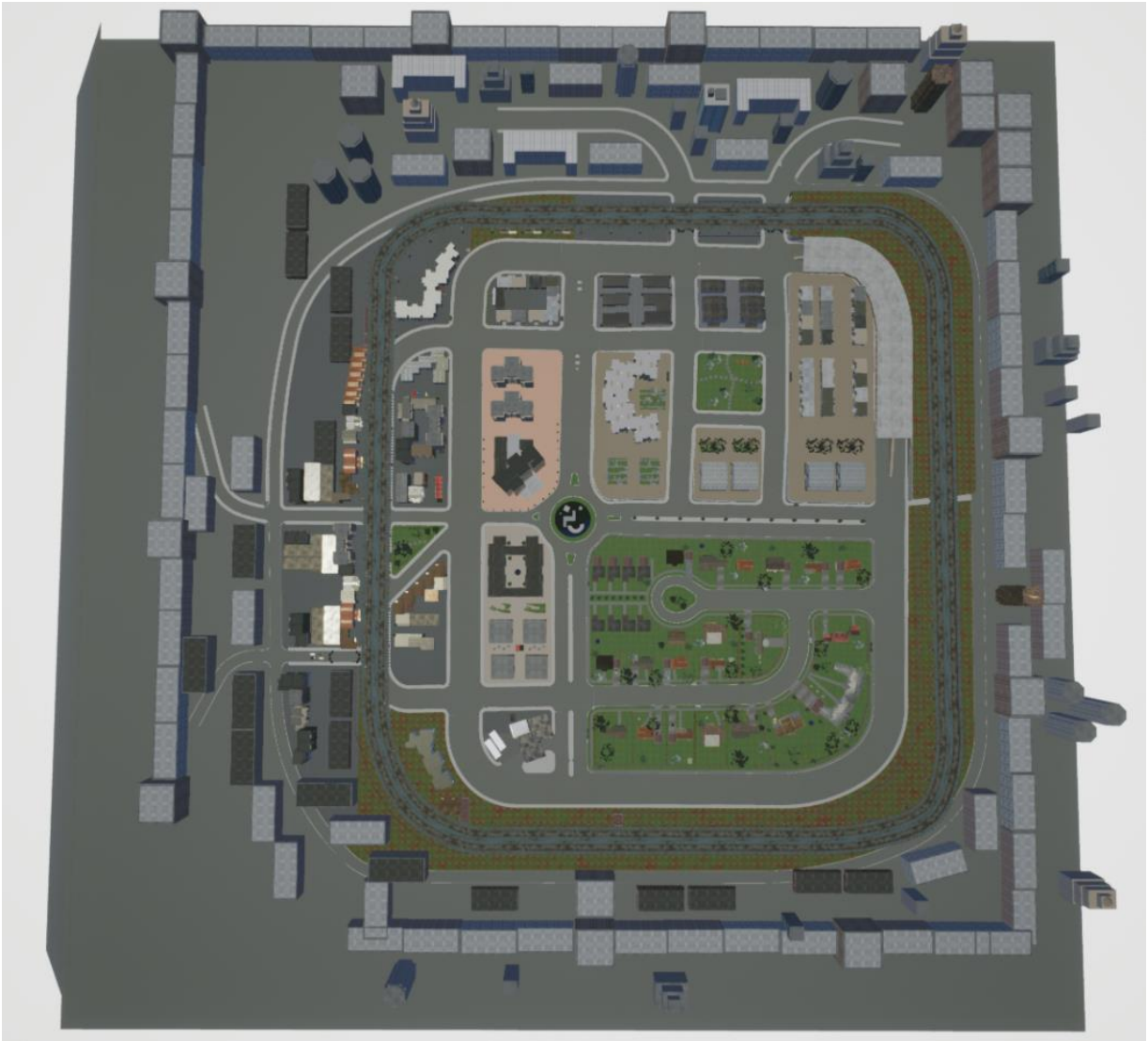

Fig. 7: Top-down view on a CARLA world

1.8.1 CARLA Environment

CARLA uses 3D models of static objects such as buildings or vegetation and dynamic objects (vehicles and pedestrians)

Static objects were placed manually and locations for spawning of dynamic objects were specified. There are several maps available, for example, Town 1 with 2.9 km of drivable roads and Town 2 with 1.4 km of roads.

The simulator is equipped with a set of sensors, such as an RGB camera, LIDAR which creates a 3D model of an environment or GNSS (Global Navigation Satellite System) used for the location of a vehicle. The car also acquires data like velocity and acceleration vector or accumulated collision impact, furthermore, the vehicle is able to measure the percentage of a car's footprint in the wrong way lines or sidewalks. Also, it can recognize the state of traffic lights and speed limits.

Apart from the pre-defined map there is also the possibility to import a road designed in VectorZero RoadRunner, but for this, it is necessary to either use special VectorZero plugin for UNREAL Engine or create a docker image of the engine which takes approximately 400 GB of disc space [22][23].

In March 2020 two new CARLA releases were issued, versions 0.9.8 and 0.9.9. The former one introduced a possibility to generate a map from OpenDRIVE[1] *.xodr* file. Example of such a map can be seen in Fig. 8

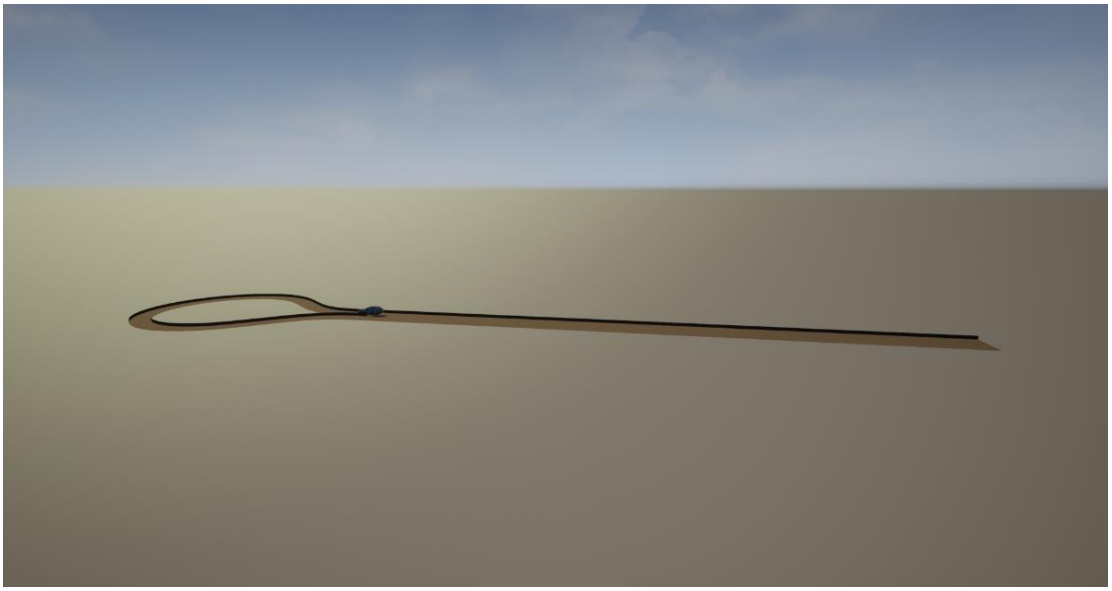

Fig. 8: CARLA road generated from OpenDRIVE *.xodr* file

[1] OpenDRIVE is an open file format in which a road networks are described. The descriptions are written in .xodr files which are created using XML. These files can be imported into various simulators such as CARLA or MATLAB AD Toolbox. For .xodr file generation e.g. RoadRunner can be used. (http://www.opendrive.org/) [20]

2 COMPUTER PHYSICS

The dynamic equation calculation for one body is not very difficult. The problem becomes more complex with more bodies in a system or especially during collisions. It can be still calculated but for simple geometries such as rectangles or circles only. If there are more complex shapes with differing frictions coefficients and densities the problem becomes exponentially more complex and "from scratch" solution can be even impossible to make.

To solve physical equations of a dynamical model (mostly motion equations and collision detection) a "physical engine" is used.

An engine is in informatics an "application core" it is a part of a software product used to solve basic, but computationally demanding tasks. The purpose of this device is to solve dynamic equations in the background of an application. There are various types of 2D and 3D engines. With a differing level of complexity. For instance, Box2D physics engine is used for the dynamic equations only, it has no rendering abilities. A more detailed description of Box2D is in the next chapter

On the other hand, there are various engines, in 3D, such as Unity or Unreal (both of which can be used to simulate autonomous vehicles as described in the previous chapter). These engines usually contain also a rendering engine that takes care of visualizing objects or it can have its own AI "sub-engine" which determines how other objects in the world (Non-Player Characters, or NPCs for short) behave.

The following chapters contain a brief introduction to some 2D and 3D physical engines, emphasis is given on a Box2D because the design of a simulator using this engine is the goal of this thesis.

2.1 Chipmunk2D

Chipmunk2D is an open-source 2D real-time rigid body physics engine. It was written in C and Objective-C but there are official bindings for Ruby and third-party interfaces, for example, for Haskell, OCaml, or Python (called *PyMunk*).

Chipmunk is used for game development for iPhone and other platforms such as Mac, Windows, or Linux.

Like Box2D, Chipmunk2D uses three types of bodies (called *cpBody*), dynamic, kinematic, and static.

Dynamic bodies are set as a default body type. They react to collisions, they are affected by gravity and other forces and their mass is a finite number.

Kinematic bodies are controlled by the user's code (not from the inside of the physics engine). They are affected by gravity and they have infinite mass so they are not affected by collisions with other bodies. They are controlled by the velocity of their movement.

Static bodies do not move in general, they can be forced to move but this results in slower computational performance. Chipmunk does not check for collisions between static bodies.

Fig. 9: Chipmunk2D logo [24]

2.1.1 Chipmunk collision detection

What concerns collision filtering the process is divided into several stages, first, there is spatial indexing where an algorithm decides which pairs of objects are the most likely to collide, and therefore they are checked for collisions. Then fast collision filtering makes first "perfunctory" collision check based on categories, groups, and types of body pair (for example shapes which are in the same non-zero group should not collide with each other).

After this, a constraint-based filtering is performed. This means that if two colliding bodies are attached and *collideBodies* property of the constraint is false the collision is ignored.

This filtering is followed by a check on whether the bodies overlap based on their geometry. A set of GJK/EPA algorithms are used for this. GJK stands for Gilbert-Johnson-Keerthi distance algorithm and it is used to determine the minimum distance between two convex bodies and EPA stands for Expanding Polytope Algorithm and is used to find the overlap of the bodies [25].

And finally, a collision handler is applied. This means that the game engine will decide how the collision should affect each of the bodies. This is either decided by built-in dynamic equation or by user-defined collision handlers similar to Box2D *contactListener()* described more in-depth in chapters 2.3.5 (general theoretical background) and 4.6 (collision detection used in this simulator) [26].

2.2 Bullet physics (bullet.org)

Bullet physics is an open-source 3D physics engine used for simulation of collision detection and soft and rigid body dynamics. It was written in C programming language but a python version called PyBullet exists, it can be downloaded simply using *pip3 install pybullet* command. The source code is hosted on GitHub. It has for example implementation of volumetric deformable objects.

Bullet can be used for creating video games or making visual effects in movies. There are also environments for reinforcement learning research.

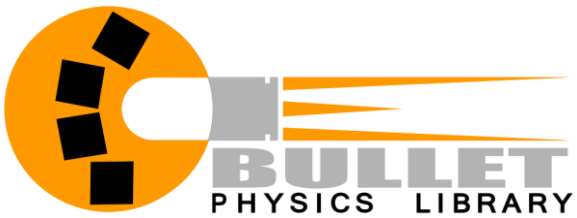

Fig. 10: Bullet physics logo [27]

2.2.1 Bullet Collision Detection

One of the key functions of any physics engines is collision detection, Bullet uses two approaches [28]:

- A posteriori ("*in the aftermath"* in Latin, also called discreet collision detection) – a simulation is done in small time steps and it is checked whether a collision occurred or not.
- A priori ("*deductively"* in Latin, also called continuous collision detection) a trajectory of each body is calculated and thus it is possible to decide whether a collision will occur.

It also uses two ways of collision detection, *ray casting* and *convex casting* the first way (as the name suggests) casts a ray from an object and measures the distance between the body and an intersection point on the other body. The latter way is more complex, it takes into consideration the whole convex body. A comparison of these ways how to handle a collision is shown in Fig. 11

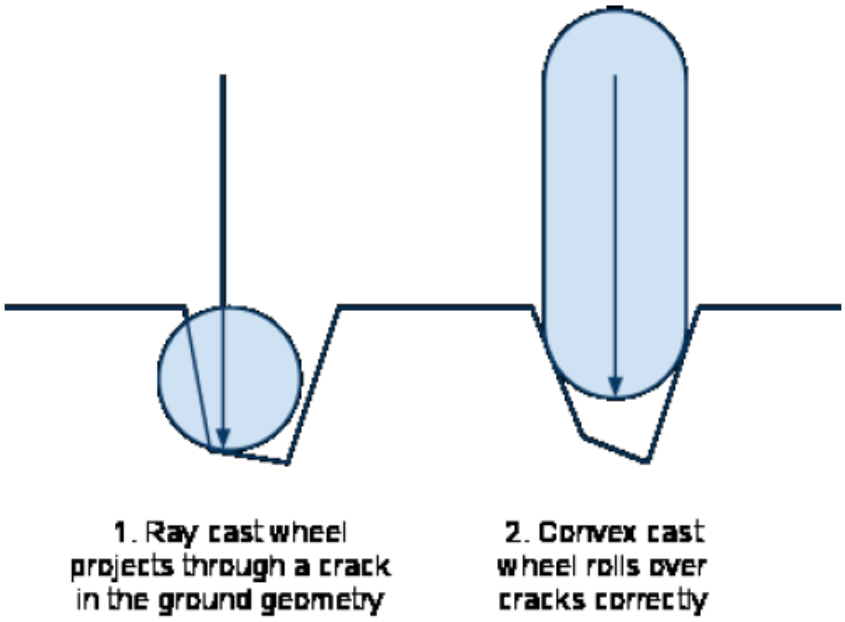

Fig. 11: Ray casting and convex casting comparison [28]

2.2.2 Bullet Dynamics

Bullet Dynamics module consists of two types of bodies, *rigid* and *soft*, these bodies have 6 degrees of freedom (translation in three perpendicular axes combined with rotation about these axes

2.3 Box2D physics engine

Box2D is an open-source operating system independent 2D physics engine introduced by Erin Catto in 2007. It is originally written in C++ but there are versions implemented to

other programming languages (in this thesis a python version called PyBox2D[2] is used). It is often used for the development of games for mobile devices. Arguably the most known game using Box2D physics is Angry Birds developed by the Finnish company Rovio Entertainment. There are different versions and the oldest version of the game has on Google Play Store only more than 100 million downloads (as of March 2020) [29].

The goal of this game is to kill green pigs by using different species of birds with varying special abilities. This can be done either by hitting them directly with the bird or by destroying their "fortresses", they have built to protect themselves. The initial state of one of the levels in the game is shown in the Fig. 12.

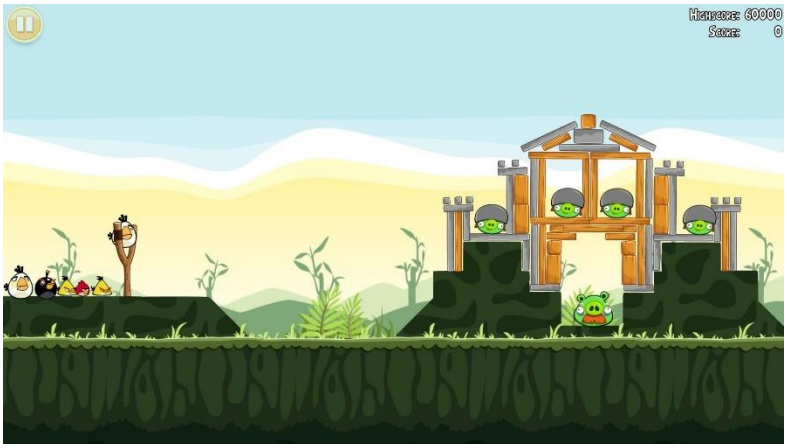

Fig. 12: Example of an Angry Birds environment [29]

As mentioned above, Box2D is a physics engine, which means it calculates dynamic equations in each step of a simulation. It is also completely graphics API agnostic. It means that the output of the engine is just data, visualisation can be done using many other toolboxes or libraries. In Python, it can be, for example, PyGame, Matplotlib, or IPython.display. Simulator designed in this thesis uses for visualizing the Box2D world PyGame library.

The engine uses meters and seconds for its calculation (in order to visualize bodies the meters must be converted to pixels usually using PPM - Pixels Per Meter - variable) and also it the uses coordinate system. It has its class for vector called Vec2

If we consider rectangle of width *w* and height *h* both parameters w and h must be divided by 2 when defining in Box2D world because the engine is considering dimensions as a distance of an object's edge to the *center* of an object

Box2D consists of four basic entities, world, bodies, fixtures, and joints, each of these entities is described below.

[2] Because the simulator designed in this thesis is written in Python, PyBox2D is used. But because Box2D is an stable name of this physical engine, further in this text the engine will be referred as simply Box2D

Fig. 13: Box2D logo [30]

2.3.1 World

World is the space where the dynamic systems are defined. It must be set up before defining any other objects. Basically, the world is a class with two attributes, gravity 2D vector, and a boolean variable doSleep which makes simulation more efficient via not assuming into the equations the bodies which are not "in use" at the moment (e.g. they are not in motion or there is no other body colliding with them). The world also calls the Step function to run the simulation.

2.3.2 Body

Body is an object with physical properties (mass point). Bodies have several properties:

- Angle
- Location
- Mass
- Velocity
- Rotational inertia
- Angular velocity

There are three types of bodies, static, dynamic, and kinematic. Static bodies, as the name suggests, does not move, it is fixed to its location, the body does not move even during collisions, it will make other bodies to move. Two or more static bodies can overlap each other.

Dynamic bodies, on the other hand, can move, a force can be applied to them. When colliding with a static body it will always retreat in accordance with dynamic properties of the collision to prevent static and dynamic bodies from overlapping.

Kinematic bodies, like static bodies, force dynamic bodies to retreat when a collision occurs. The difference between static and kinematic bodies is, that kinematic bodies can move.

Bodies can also have a *sensor* attached to them. The sensor is a property that detects whether or not are two bodies overlapping (not to be mistaken with contact listener which is used for collision detection). In vehicle simulator, it can be used for creating some areas with different traction (for example roads in a vehicle simulator can represent asphalt, but there can be various areas to represent gravel or puddles, these would be represented with static bodies with sensors, so a dynamic body – a car – could overlap it and also the body would affect the car behaviour like in real-world situation).

2.3.3 Fixture

A fixture represents a connection of geometry to the mass point. For a body, it is impossible to know how it will behave when it collides with any other object or how does it look like until a fixture is added to a body.

Fixtures have four main properties:

1) Shape
   Body geometry – it is a vital property of every fixture, it is used to check whether a collision with other bodies occurred. There are three basic types of shapes, Polygon, Circle, and Edge. The first two have volume, the third does not. It is also a reason why two Edge shape cannot collide with each other because Box2D requires that at least one of colliding bodies has a volume.
2) Restitution
   It has values from 0 to 1 and it measures how much of energy will be conserved when a collision occurs.
   0 means, that when two fixtures collide all energy of the "bounce" will be conserved
3) Friction
   The meaning of it is the same as in the real-world
4) Density
   Meaning is also the same as in the real-world. It is used to calculate a mass of a body, by multiplying the density with the area of a body.

2.3.4 Joint

Connecting bodies, there are several types of joints

- Revolute – bodies can rotate about a common point
- Distance – keeping a fixed distance from bodies
- Prismatic – fixed relative rotation, sliding along an axis is allowed
- Wheel – a combination of revolute prismatic joint, used for side-view car suspension modelling (the simulator designed in this thesis used top-down view, therefore, it uses a different type of joints, creating a car is described in chapter 4.2
- Weld – holds the bodies
- Pulley – holds points on each of two bodies within a given distance from a third point in the world
- Friction – reduces relative motion between two bodies
- Gear – controls two other joints, either prismatic or revolute so the movement of one joint affects the other
- Mouse – pulls a point of one body to a location in the world
- Rope – Constraints a maximum distance two bodies can be moved apart

All Joints have common properties when they are defined, bodies which are going to be connected had to be defined (bodyA and bodyB) also their anchor or relative anchor (anchorA and anchorB or relativeAnchorA and relativeAnchorB) anchors define a point in the world in a global coordinate system and relative anchors define points in a coordinate system relative to the body (for example relativeAnchorA = (0,0) means, that the anchor point is defined in the centre of the body).

There can be other parameters depending on the exact joint type, for example, revolute joint can have upperAngle and lowerAngle parameters which constraints limits of motion or rope joint needs the maximum distance two bodies can be apart.

2.3.5 Collision detection and filtering

To handle collisions the world is equipped with so-called “contact listener” which gives the user the opportunity to see whether any collision occurred, and which two bodies collided. To filter the collisions, it is necessary to ad masks and categories to each type of body. Collision filtering in the simulator designed in this diploma thesis with examples of code is shown in chapter 4.6. And simple user-defined class with contact listener is shown in a section of code below

```
1. class myContactListener(b2ContactListener):
2.     def __init__(self):
3.         b2ContactListener.__init__(self)
4.     def BeginContact(self, contact):
5.         # Defines what happens when a contact between two bodies
    occur
6.         # (for example, print("There is a contact")
7.         pass
8.     def EndContact(self, contact):
9.         # Defines what happens when two bodies stop colliding
10.          pass
11.      def PreSolve(self, contact, oldManifold):
12.          pass
13.      def PostSolve(self, contact, impulse):
14.          pass
```

2.3.6 Ray Casting

Like Bullet engine, Box2D has a function of ray casting, a ray is a straight line and it is possible to find out whether the ray crosses any fixture.

The direction of a ray is given by a two points p1 and p2, it is also possible to set a distance, how far along the ray it should be checked for intersection, it is also possible to find a normal to the surface on which the intersect point lies. Ray casting can be used for example in a shooting game where the ray can split an object it hits.

2.3.7 Setting up Box2D for Windows

Box2D is a standard Python library, so it can be downloaded using pip or conda, simply by writing command pip install Box2D or conda install Box2D in respective command lines. Using anaconda and conda install command, the environment will find any other libraries (called also dependencies) and download them automatically, if necessary (i. e. NumPy). On the other hand, using pip installer the libraries needed must be downloaded manually, it also might be necessary to download and install them not by using pip but to download and manually install so-called .whl file.

This problem occurred when installing Box2D for the purpose of this thesis, a .whl file for Box2D had to be downloaded and then installed via command line.

3 ARTIFICIAL INTELLIGENCE USED IN AUTONOMOUS VEHICLES

This part of the thesis can be divided into two main sections, artificial intelligence used in autonomous vehicles in general (especially object detection, or pathfinding algorithms) and artificial intelligence which is supposed to be used in the simulator which design is in this thesis described

3.1 Artificial intelligence in general

As mentioned above, there are two main fields of AI for autonomous vehicles, first is object detection (pedestrians, other vehicles, trees, buildings or both horizontal and vertical traffic signs) and pathfinding and following algorithms.

3.1.1 Machine learning

It is a subset of artificial intelligence which is focusing on algorithms and methods in order to allow a computer system to "learn". Learning or training is such a change of the system's internal states that will improve the system's ability to adapt to changes in the external environment.

In other words, computer systems based on machine learning improve themselves automatically through some kind of experience. This field lies at the intersection of computer science and statistics [31].

The main problem with machine learning is that it is very demanding what concerns time and computational power. That is the reason why the rapid development of ML algorithms began in the last decade or so and not sooner, computer systems in the past were simply not powerful enough to perform complicated computations in a reasonable time.

Nowadays the usage of the machine learning algorithms is very wide, it differs from targeted advertising through stock price predictions, warning about traffic jams, and recommending alternative routes in a navigation system to cancer detection [32].

3.1.2 Pathfinding and following algorithms

Before an autonomous vehicle begins to move it must know its starting and desired location, from which it will calculate the optimal path. It uses data obtained from GPS sensors and map files such as Google Maps or OpenStreetMap.

In order to achieve more precise localisation, the vehicle can also use data from other, than GPS sensors, e.g. data obtained from LIDAR and process them by Kalman filter or similar control algorithm. To finding optimal path several types of algorithms can be used, for instance, A* or Dijkstra [33]. The algorithms can also implement various inputs such as finding either the fastest or shortest route or use toll-free routes only [34].

3.1.3 Object detection

This consists of detecting a very wide field of object types, Pedestrians, animals, other vehicles, bicycles, buildings or traffic signs can be detected. For this purpose, usually, some kind of machine learning is used. For horizontal traffic signs (road lines) image processing algorithms such as colour adjustment and colour crossing detection can be used.

As a very good example, a vertical traffic signs detection can be used. There are several ways how to approach this problem, one of which is using Viola-Jones Detector which uses machine learning algorithm AdaBoost [35].

3.2 Artificial intelligence for this simulator

The goal of this thesis is to design a simulator that will be later used for mostly educational purposes for machine learning algorithms. The most suitable sort of algorithms seems to be *reinforcement learning* and its varieties *Q-learning* and *Deep Q-learning*.

These algorithms can be demonstrated using platform OpenAI and its product OpenAI Gym [36].

OpenAI is a for-profit organisation founded by Elon Musk, Sam Altman, and group of other investors in 2015. The main goal of the company is to conduct research in artificial intelligence. Musk and partners invested over 1 billion USD to the venture and in 2019 another 1 billion dollars was invested by Microsoft Corporation.

An interesting example of their work can be that in 2018 they trained robot hand to be able to manipulate a cube in order to face the requested side upwards. They used reinforcement learning and its variety called domain randomisation. The system developed a human-like behaviour like finger pivoting and sliding of a cube without being programmed to do so [37].

Another interesting achievement was made on April 19th, 2019. On that day team OpenAI 5 defeated human world champions in very complex PC game Dota 2. The agents were trained using large scale deep reinforcement learning.

Fig. 14: OpenAI logo [38]

3.2.1 OpenAI Gym

It is a product of the OpenAI organisation for developing reinforcement learning algorithms. It is open source and source code is hosted on GitHub. It is written in Python and it can be installed using simple command pip install gym.

The Gym contains several tasks (called environments) from simple text-based such as Frozen-Lake, classic Atari games like Ms. Packman, classic control tasks, for instance swinging up a double pendulum to make it inverse pendulum, robotic tasks like a robot hand model mentioned above and also some environments based on Box2D engine. There are environments with for example a bipedal robot that is learned to walk either on flat ground or ground with obstacles, lunar lander or top-down racing car is learned to make a round on a race circuit.

3.3 Reinforcement learning and its varieties

Reinforcement learning is arguably one of the leading algorithms for autonomous vehicles simulation and modern artificial intelligence in general. For instance, both CARLA and Microsoft AirSim, mentioned in chapter 1.1 and 1.8 respective can be used for reinforcement learning or its varieties for autonomous vehicle simulations. Also,

OpenAI, described in the previous paragraph, is most suitable for reinforcement learning (or Q-learning) applications.

Reinforcement learning is based on Markov decision processes and it is trying to take actions in order its total *reward*.

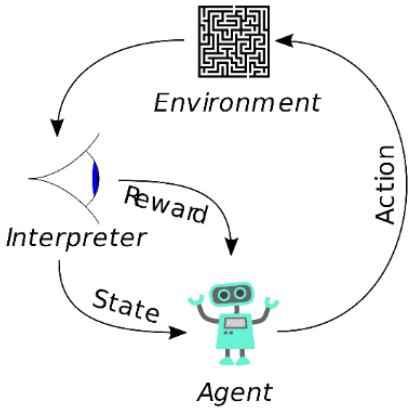

Fig. 15: Usual diagram of a reinforcement learning episode [40]

3.3.1 Imitation learning

Imitation learning can be used to train a model that maps perceptual inputs to control commands, for example, it can relate images obtain from camera to acceleration and steering [41].

3.3.2 Q-Learning

It is a model-free reinforcement learning algorithm. Its aim is to create so-called *Q-Table* which tells an agent which action it is supposed to take under what circumstances. The table is usually initialized with either zeros or random numbers and after training, via formula (1) the table is filled with Q-values. The initialized and filled Q-table can be seen in Figure 16.

$$(1)\ Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t - a_t) \right)$$

Where:

- a_t – action selected by the agent
- s_t – old state of the environment
- s_{t+1} – new state of the environment
- γ – discount factor ($0 \leq \gamma \leq 1$), it values earlier received rewards higher than the later rewards
- r_t – reward for moving from the state s_t to the state s_{t+1}
- α – learning rate ($0 \leq \alpha \leq 1$)

Below is simple pseudocode that implements a Q-learning formula on an OpenAI Gym environment.

```
1. Import gym
2. env = gym.make("EnvironmentName")
3.
4. action_space_size = env.action_space.n
5. state_space_size = env.observation_space.n
6.
7. q_table = zeros((state_space_size,action_space_size))
8.
9. done = False
10.  rewards_current_episode = 0
11.
12.  for step in range(MAX_STEPS_PER_EPISODE):
13.          exploration_rate_threshold = random(0,1)
14.  if exploration_rate_threshold is bigger than exploration rate:
15.          action = argmax(q_table(state,:))
16.  else:
17.          action = env.action_space_sample
18.
19.  new_state, reward, done, info = env.step(action)
20.  Update Q-Table:
21.  q_table(state,action) = q_table(state,action) * (1-
   learning_rate) + \
22.  learning_rate * (reward + discount_rate *
   max(q_table(new_state,: ))
23.          state = new_state
24.  rewards_current_episode += reward
25.  if done == True:
26.  break
```

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	327	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Fig. 16: Q-table originally initialized (in this case) with zeros and then after training filled with Q-vaules[42]

As mentioned above, Q-table is used to decide which action the agent should take when the environment reaches a given state. It takes the action which has the highest Q-value. For example, when the environment described by the Q-table in Fig. 16 reaches state indexed 499 the agent takes action called “West (3)” because its Q-value (29) is the maximum of all possible action within that state.

4 SIMULATOR DESIGN

According to the diploma thesis assignment, the goal of this thesis is to design an environment using Box2D physical engine which will later be used for educational purposes. It should serve as a platform in which the students will apply their machine learning (especially Q-learning) algorithms.

The advantage of this simulator in comparison with for example CARLA is, that CARLA repository needs approximately 2,5 GB of data to run before any artificial intelligence is applied. CARLA also takes a considerable amount of computational capacity of personal PC, mainly for graphics rendering, this can result for example in quick overheating of a laptop.

Simulator designed in this thesis needs for its run a Python interpreter such as Visual Studio Code and nothing more. Also, the size of this script is in order of hundreds of kilobytes at most, that means that it takes only a fraction of memory needed in comparison to CARLA, furthermore, because the environment of this simulator is much simpler than the one in CARLA (consists of only three types of entities which will be described below) The rendering and computational capacity needed is much lower as well.

For this simulator, a Box2D world was created, it consists of three entities, a car, pedestrians (these two are dynamic bodies), and buildings (static bodies).

The car is the main entity on which a machine learning algorithm will be applied (car generation process is written in chapter 4.2).

Even though by default the simulator environment contains only one car, which can be controlled either manually or as an agent it is very easy to generate almost any number of other cars, thanks to creating cars as instances of class Car.

Pedestrians are represented by circles of a given radius and they are dynamic bodies (more detailed description is in chapter 2.3.2), buildings are made as static bodies of a rectangular shape with user-given dimensions. To make the city grid look more like an actual city with various width of streets and several sizes of buildings there are three different building footprints. Buildings of different dimensions are placed in a city grid randomly.

4.1 Creating a Box2D world

The world is created using b2World object which needs two keyword arguments (or kwargs for short), gravity and doSleep, which reduces the amount of calculations needed by not simulating bodies which come to rest (e.g. do not move), other kwargs such as *contactListener* does not have to be passed.

In case of this simulator a Top-Down view is needed so gravity is set to (0,0). Box2D world is thus instantiated using following line of code:

```
1. GRAVITY  = (0,0)
2. box2world = b2World(gravity = GRAVITY, doSleep = True,
   contactListener = myContactListener)
```

The contact listener is a class that detects collisions of bodies in a Box2D world and will be described in chapter 4.6.

4.2 Creating a car

Car is defined using two classes, *class Wheel* and *class Car*, in class Car four instances of a class Wheel are created, front wheels are connected to the car using *Revolute Joints* and rear wheels use *Weld Joints* because they do not move so this is the most suitable joint that Box2D offers. The designed and rendered car is shown in the Fig. 17

Fig. 17: Top-Down car rendered using PyGame

4.2.1 Cancelling lateral velocity

Because Box2D is developed for 2D Side-View dynamic situations and this simulator uses Top-Down view it is necessary not only to set gravity to zero but also "cancel lateral velocity" in every step of a simulation. What a lateral velocity is and how to find it is shown in Fig. 18.

The lateral velocity is cancelled by applying impulse, which vector has the same size but opposite direction as lateral velocity vector. If we cancelled lateral velocity completely the car would move like a train on rails. This is not very realistic behavior and it can be corrected by setting a maximum value of cancelling impulse. If lateral velocity is bigger than the pre-set maximum value, it will not be cancelled completely, and the car will skid.

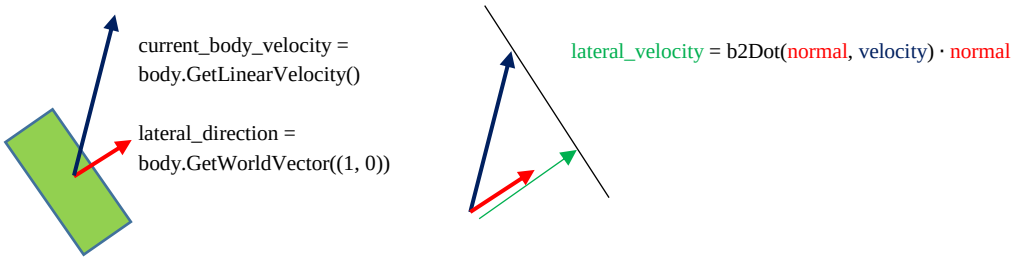

Fig. 18: Lateral velocity of a body (modified to Python syntax from [43])

4.2.2 Controlling a car

For this purpose, a method *action(self, key)* was made in a *class Car*. This method represents the action space that a machine learning algorithm can take.
Actions are indexed from 0 to 3, actions 0 and 1 mean go forward and go backward respectively and actions 2, 3 stands for go left and go right respectively. Actions 0 and 1 use for achieving and then keeping constant speed a *b2Body.ApplyLinearImpulse()* command, this is the most realistic tool for moving and achieving a constant speed in Box2D. What concerns turning, two ways of controlling a tire were considered:

1) Using *b2RevoluteJoint.motorspeed* property, this is physically more realistic but, on the other hand, it is much more difficult to achieve precise steering.

2) Direct setting of a turning angle via *b2RevoluteJoint.upperLimit* and *b2RevoluteJoint.lowerLimit* properties. This is a simpler and more precise way but an unrealistic way concerning physics.

Also, a decision what to do when the car reaches given boundaries of a map had to be done, also two ways were considered

1) Make a car wrap-around effect, similar to pedestrians, that means when the car body position exceeds given limit it will automatically appear on the other side of a map
2) Destroy the car and create a new one on a random position

Because this simulator is supposed to serve as an environment for machine learning algorithm development and especially reinforcement learning algorithms option no. 2 was used and destroying the car is one of the possible ways how to terminate an episode. Making the simulator compliant for this purpose is described in chapter 5.

The way how to check whether the car reached the map boundaries was realized via checking the position of the centre of the car in each time step and to destroy the car whenever it exceeds given limits (both horizontal and vertical limits were set). Before this solution, a different approach was considered and it consisted of defining an edge chain body that would be used as a border, pedestrians would not collide with the body but when the car hits the border it will be destroyed. But edge chain bodies have by Box2D definition no volume and collision filtering when one of the colliding bodies have edge shape is more difficult, checking car position was found to be a simpler and more effective solution.

4.3 Creating buildings

As mentioned above, buildings are created as static bodies of a rectangular shape with three different sizes in order to make the map look more like an actual city. Buildings are generated as instances of *class Building()* which takes as input arguments the building size and building position.

The building shape must be rectangular by default, with user-defined lengths of sides but it is possible to easily rewrite the class to allow more complicated shapes. By default the shape – which is a fixture – is defined using function *getFixture()* within the *__init__()* method of a class. This function takes several arguments and one of these is defined using *shape=polygonShape(box=(self.shape))*. The *box=(self.shape)* ensures that the building shape will always be rectangular. If user want to make more complicated shapes it is possible to delete the "box" part change for a list of up to 8 tuples which represent tops of the polygon. The polygon, also, must be convex, Box2D cannot work with concave polygons (although this can be solved by splitting a concave polygon into a set of convex polygons)

Using this class building it is only possible to create polygon-shaped buildings (default setting) only or circle-shaped buildings only, it is not possible to mix these shapes in a single class. This is caused by the *shape* argument in fixtures definition, there can be either *polygonShape*() or *circleShape()* with a given radius, it is not possible to combine them. There are two possible ways how a user could solve this issue if he wanted to. Either he could define two similar classes named for example *PolygonBuilding()* and *CircleBuilding()* or he might define an argument named for example *building_shape* and within the *__init__()* method an if-statement would be defined which would decide, based

on the value of the *building_shape* argument which shape the building would be a polygon or a circle.

Anyway, the shape of the building has a very small influence on the performance of the simulator so this issue was not solved in the code and all the buildings are by default rectangular. The purpose of the upper paragraph is to give a future user a hint on how to define a more complex city grid if he or she wanted to.

4.4 Creating a city grid

As stated in chapter 4.3 the buildings are rectangular and there are by default three sizes of the buildings.

There have been two city grids designed, bigger and smaller. For both a list of buildings position is defined and on each of these positions, a building is placed in a for loop. The smaller one has a shorter list and so it consists of four buildings only with identical dimensions. There is a smaller number of pedestrians as well (by default six). The larger map is defined by a longer list of positions so a bigger number of buildings are generated.

Visualization is made using PyGame library and the bigger map was tuned in order to fit Lenovo ThinkPad E560 screen (when the PyGame screen is set to full screen). Using different monitor the buildings will not fit the screen properly (either there is a blank white space around the buildings or there might be only a part of the buildings in the last row and column visible). When displaying the smaller map, there is no need for maximizing the window, everything important occurs in the visible screen.

The complete world with the bigger map rendered using PyGame library is shown in Fig. 19.

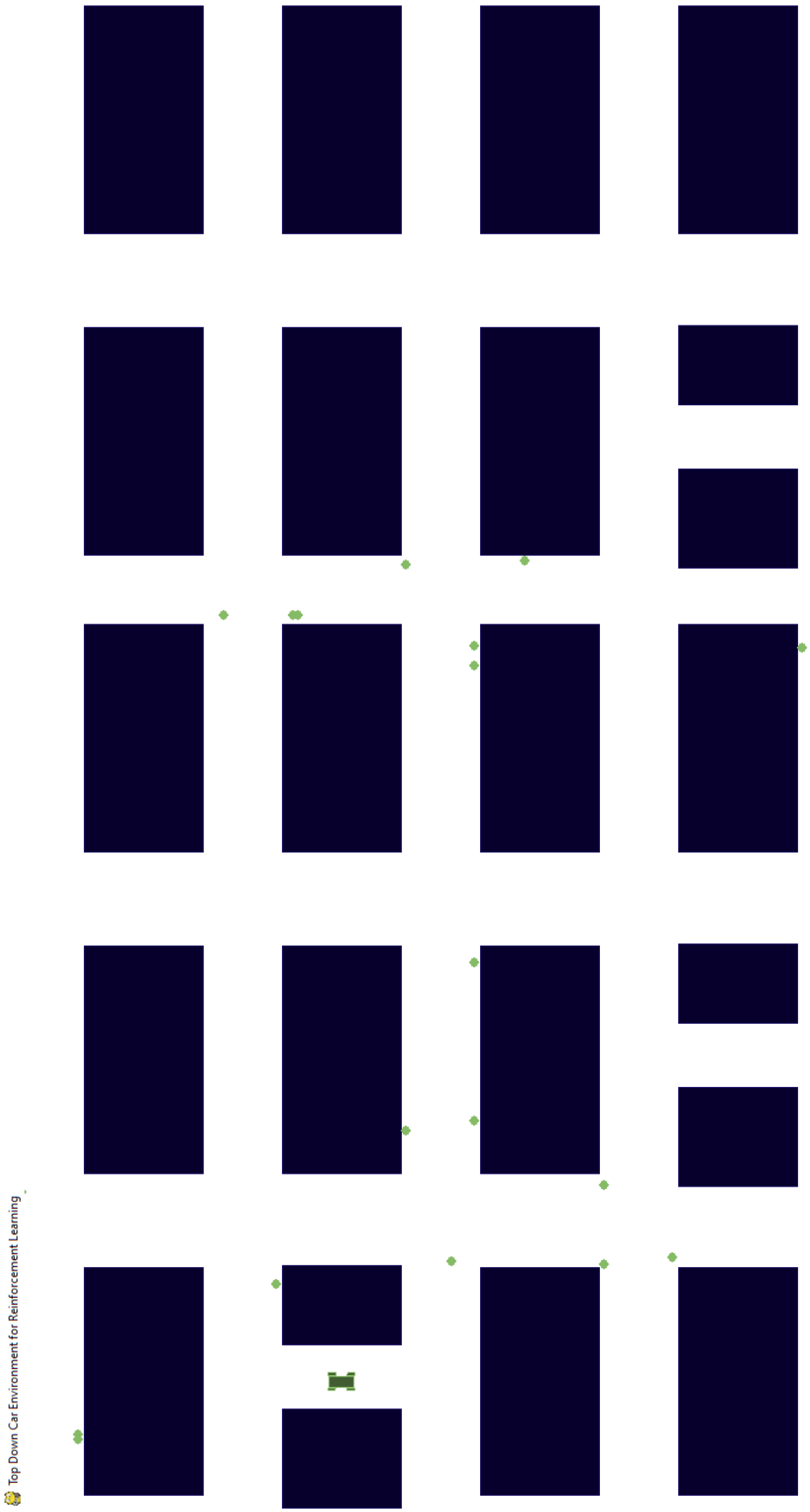

Fig. 19: Complete world with a car, pedestrians, and buildings (bigger map)

The reason behind creating two maps is that according to the last requirement from the diploma thesis assignment is to test the simulator functionality. When the smaller map is used with a smaller number of pedestrians the functionality testing is easier, the computational time demand is much lower. But when the bigger map is used the working principle of this simulator should not change at all. The only difference should be that it would be computing more slowly.

4.5 Generating pedestrians and algorithm of their movement

Pedestrians are generated in a finite loop (a for loop), in each iteration a random building is chosen, then a pedestrian is generated within the footprint of the building, because buildings are static bodies and the pedestrians are pushed away from the footprint and they begin to move, the direction of the movement is given by their position when they are forced to move away from the footprint.

Movement of a pedestrian is realized by the actual setting of its velocity using *b2Body.__SetLinearVelocity()* command. It is the simplest way of achieving the desired speed and controlling movement. The velocity is assigned when an instance of a Pedestrian class is created via attribute *ped_velocity*.

By default, their velocity is set to a random value from a given interval. This value is then assigned to each pedestrian when it is created. The limits of the interval are set to quite unrealistically high values. The reason behind this is that the map is quite large and if the pedestrians were slow their movement could have a small impact on the behaviour of the vehicle.

In case of the pedestrian reaching edge of a given environment, a wraparound effect was implemented. That means that after disappearing beyond one of the edges the body will immediately reappear on the other side of an environment.

If pedestrian's position is between the upper and lower edge of a building it means that they were aligned next to a vertical side of a building and the pedestrian starts to walk up or down

If its position is between the left and right edge of a building it means that the body was positioned next to a horizontal side of a building and it starts to walk left or right

To each pedestrian an integer is assigned. The number is either +1 or -1 It is used to decide which direction the pedestrian moves. That means that one group of pedestrian is moving left, the other is moving right, the third group is moving up and the fourth is going down. This ensures that the movement of the complete set of pedestrians is more realistic.

If the pedestrian walks into a given area near any edge of a building than it will either keep moving or it will change its direction by 90 degrees, the decision depends on the exact location of a pedestrian.

If the pedestrian's position does not change in two consequential steps (that means the pedestrian hit edge of a building) he should start to move in a direction that is perpendicular to its previous direction.

During testing of this functionality, an error was detected in which the pedestrians do not move at all, but because this happens quite scarcely and to a few of the pedestrians there is no need for fixing it, this can represent, for example, a pedestrian who stopped and is watching inside a store.

4.6 Collision filtering and handling

For collision filtering a contact listener was set as described in chapter 2.3.5. In each step, it is checked whether a collision between a given pair of bodies occurred. The pairs are (given as instances of its classes):

- Car & Pedestrian
- Wheel & Pedestrian
- Car & Building
- Wheel & Building

Collisions between instances of class Pedestrian and class Building are occurring but they are not important for the functionality of the simulator, which means that they are allowed (a pedestrian collides with a building), but no callback of such collision is needed and it is solved via the Box2D engine itself.

On the other hand, collisions between pairs from the list above are important for the simulator and therefore they are filtered and then when such collision occurs and it is used as a way how to terminate reinforcement learning episode (as described below)

4.6.1 Groups, masks and categories

In order to achieve desired collision filtering during initialization of an instance of each class a filter had to be attached to instance fixtures. The filter needs three types of information, which category the instance is a member of, which group it is in, and which mask it has.

For setting category and mask bits integer values which must be a power of 2 numbers are used, categoryBits are the main object to test the collisions on, maskBits are used to determine which objects can the tested object collide with if it is needed to check collisions for more than just one other type of objects it is possible via adding categoryBits of all the requested objects.

Finally, *groupIndexes* are set. They determine layers and whether the fixtures within the same layer collide or not.

If the layer has a negative number than all the fixtures will not collide with each other.

An example of setting a filter with categoryBits, maskBits and group indexes is shown in a code example below (setting a fixtures and a filter in class Car):

```
1. self.body_Fixtures =
   self.body.CreatePolygonFixture(shape=b2PolygonShape(box=(2.189 /
   2,4.897 / 2)),
2.                                density=2810,
3.                                friction=0,
4.                                filter=b2Filter(
5.                                   categoryBits=CAR_CATEGORY,
6.                                   maskBits=PEDESTRIAN_CATEGORY +
   BUILDING_CATEGORY,
7.                                   groupIndex=CAR_GROUP))
```

4.6.2 Setting a contact listener

As described in chapter 2.3.5 Box2D filters contacts using so-called *listener* which is set up as a class which instance will then be passed into a world definition. The class by default has five methods, __init__, BeginContact, EndContact, PreSolve, PostSolve For

needs of this simulator only __init__ and BeginContact were used, also an auxiliary method HandleContact was written.

The HandleContact method finds out which fixtures collided, then which body belongs to the fixture and finally, it finds out which class is the body instance of.

Based on this information a decision is made what to do, if the colliding bodies are instances of class Car and class Pedestrian the world is destroyed, the negative reward is awarded, and Q-learning parameter done is set to *False*. The same happens when the colliding bodies are instances of classes Car and Building, the only difference is that the reward has a different value.

4.6.3 Destroying bodies and defining a step function

To destroy a world three functions or methods had to be written, first is *_destroy()* which controls parameter *destroy_flag* of a car which is by default set to *False*. When a contact is detected in ContactListener method *HandleContact()* the function is called and the flag is set to True. Then function *step()* is called, normally this function controls whether the car moves from the map and also is taking care of calculating dynamic equations for each step by built-in Box2D method *b2World.Step().* But when the *destroy_flag* is set to *True* dynamic equations calculation is interrupted and a for loop is activated. The loop iterates over all b2World bodies and applies the built-in Box2D method *b2World.DestroyBody()* which takes care of destroying the body.

The reason why the time step counting and dynamic equations computing has to be interrupted is that it simply cannot be done simultaneously. This is given by the inner algorithms of Box2D and trying to do so will result in assertion error *"IsLocked == False".*

After this, a function called *reset()* is called. This function creates a new set of pedestrians and buildings and also a new car and sets the car's *destroy_flag* parameter back to *False*

4.7 Creating a simulator API

According to the thesis assignment, it is demanded to create an API that will serve as a user interface. This will later be used to train machine learning algorithms (especially reinforcement learning algorithms).

The ideal output of this simulator design is to create an environment using Box2D physics engine, in which the user only sets basic parameters such as a number of episodes, which episode he or she wants to render, which artificial intelligence method he or she wants to use (that means for example that the user applies the Q-Learning formula) and so on. And the simulator will work as some kind of "black-box".

The code is open which means that the user can make any choices he or she deems appropriate (e.g. that the user doesn't want to train the car to leave the grid but to maintain a minimum velocity for given amount of time steps). The simulator has to have some preset default parameters and settings but it also has to be possible to allow the user to rewrite it easily.

This is achieved for example via using object-oriented programming (or OOP for short). The OOP allows quite an easy way to apply changes by adding or removing or editing just one or a few methods/objects.

As an example of this can be said that if the user wants to change what happens when a collision occurs all he or she has to do is to edit one method within *ContactListener()* class.

Another example of how the simulator can be user-friendly is that there is a built-in parameter *manual_mode* which is by default set to False, this means that the user cannot apply any actions to the car manually, but he or she has to achieve this via some automated way such as Q-Learning. When the parameter is set to True, the car in the simulator can be controlled using WASD keys on a keyboard.

4.7.1 Setting the world to be suitable for a machine learning environment

In order to use this environment for the education of machine learning (especially Q-learning) education the simulator needs several functionalities and ways how to handle scenarios. These functionalities include:

1. Environment initialization
   This means simply to create the environment, in case of this simulator a set of entities consisting of buildings, pedestrians and a car is created
2. Episode termination
   The episode is finished when the car either hits the pedestrian or hits the building or its position is out of the defined area or the episode takes too much time to evaluate (for example a limit that the episode must be terminated after *x* time steps was met). These scenarios set the parameter *done* to *False* this means that the car did not meet the expectations.
   The parameter is set to *True* when the car drives out of the map,
3. Environment reset
   When one of the scenarios for episode termination is fulfilled all bodies in the Box2D world are destroyed and they are replaced by a new set of bodies according to the rules described above. That means a pseudo-random number of buildings, given amount of randomly distributed pedestrians and a car in random position is generated.
4. State space definition and action space
   There are several ways of defining the state space, in for purposes of testing two were defined, they are described in chapters 5.1.1 and 5.1.2
   The car can make one of four actions, it can either go forward or backward or it can turn left or right.
5. Effect the environment
   There must be a way how to force the agent to take the action and affect the environment
6. Reward system
   A negative reward is given to the car when the episode is terminated with the parameter *done* set to *False*. That means the car hit either a pedestrian or a building both of these situations are awarded by a preset reward. When the car finishes its task, that means for example it was able to drive out of the map the reward is equal to zero. This is given by the mathematics behind Q-Learning negative rewards are more useful than positive ones that means that the “positive reward” (or reward for finishing the task successfully) can be zero.

Because the backbone of the Q-Learning is repeating the tasks in form of *episodes* for a very large amount of times (usually tens to hundreds of thousand times) it is time-saving not to render each one of the episodes. This is one of the advantages of using physical engine such as Box2D because it completely graphic API agnostic. This means that episodes can run in the background and the user can render only a limited number of them

or no episode at all. It is possible to set the algorithm in a way to generate every *n*-th episode (for example every 3000^{th} or 1500^{th} episode).

Each episode is given a maximum number of time steps it can take. If the car did not get out of the map within the given limit the episode is terminated.

This can be considered as a positive or a negative situation. If a positive approach is taken it can mean that the car finished the task and hit neither pedestrian nor building. If this is considered a negative situation it means that the car was not able to find a way from the city grid. The positive or negative consideration depends on the user which approach he or she wants to take (taking this as a negative phenomenon makes solving the task much harder).

5 TESTING THE FUNCTIONALITY OF THE SIMULATOR FOR Q-LEARNING

It is important to note that according to the diploma thesis assignment the goal of the thesis is to design a simulator that will be in the future used for the development of artificial intelligence algorithms. This means that it is not necessary to develop a working machine learning algorithm but it is needed to prove the simulator to be ready to use for such purpose. For example, that the car is taking legal actions, that the episode is terminated after a given condition is met, and so on.

5.1 Q-Table definition

As can be seen in Fig. 16 the Q-table describes a relation between states of the environment and actions the agent should take. The Q-values are found out via training and formula (1) is used for that. When the environment reaches a given state, the action with the highest Q-value out of all possible actions is taken as described in chapter 3.3.2.

5.1.1 State-space defined only via pedestrian angles

For purposes of testing this simulator the state space is defined for *n* nearest pedestrians and each pedestrian can reach either state 0 or 1. The number *n* can be chosen arbitrarily and can be lower or equal to the total number of pedestrians in the environment. In this case $n = 5$. State 0 means that the angle between the pedestrian and longitudinal axis of the car is lower than a given threshold, state 1 means that the angle is larger than the threshold. The threshold can be chosen arbitrarily as well, for purposes of testing the chosen value is $30\,° \left(\frac{\pi}{6}\,\mathrm{rad}\right)$. The agent can take *m* actions (in this case $m = 4$, the car can either go forward or backward or turn left or right).

In principle, the state space is just a binary representation of numbers from 0 to - in this case - 31. There are several ways how to do that in Python, for example via using python built-in libraries or defining a function using recursion. In computer science, recursion is a method when the function is called again before the previous calling was finished (in other words “the function is called within the same function”) [44].

The former way can look like this:

```
1. import itertools
2. itertools.product(*[[0, 1]] * n) # n is the power of 2 number we
   want to represent
```

The latter, which is also used in this simulator, using recursion

```
1. def state_space_generator(n): #n has the same meaning as in the
   previous snippet
2.   if n == 1: return [[0], [1]]
3.   recursion_space = state_space_generator(n-1)
4.   return [[0] + c for c in recursion space] + [[1] + c for c in
   recursion_space]
```

This way is then just simply called this way:

```
1. state_space = state_space_generator(NEAREST_PEDESTRIANS)
```

The total amount of cells with parameters in the Q-table than is calculated as follows:

$$X = 2^n \cdot m = 2^5 \cdot 4 = 128$$

Where:

- X – total amount of Q-values
- n – number of closest pedestrians
- m – number of actions the agent can take.

And finally the Q-table is initialized using this code:

```
1. qtable = [] # "Action space"
2. for i in range(len(state_space)):
3.  qtable.append(np.random.uniform(low=-5
   , high=0, size=(len(actions),)))
```

In other words the code above fills the "action space" with random Q-Values from -5 to 0 the method random.uniform() is a pseudo-random number generator with uniform distribution.
So when initialized the Q-table can look like this:

States of nearest pedestrians					Possible actions			
ped0	ped1	ped2	ped3	ped4	a0	a1	a2	a3
					Go Forward	Go Backward	Turn Left	Turn right
0	0	0	0	0	-3,376	-2,211	-3,101	-3,991
0	0	0	0	1	-0,869	-2,541	-3,944	-1,114
0	0	0	1	0	-4,330	-4,067	-1,841	-3,505
0	0	0	1	1	-2,058	-0,700	-2,894	-2,225
0	0	1	0	0	-0,728	-2,890	-3,194	-2,923
0	0	1	0	1	-3,222	-0,744	-0,568	-1,758
0	0	1	1	0	-4,120	-2,583	-0,139	-1,188
⋱					⋱			
1	1	1	1	1	-4,720	-3,749	-3,344	-2,041

Tab. 1: Q-Table when initialized

Even though this state space representation is not the final version, the source code which is given to this thesis as an appendix is using the second type of representation which is described in the following chapter, the description of this state space can be a good idea for the possible future user of the simulator. For this reason, the description of how this state space was created in this chapter was written.

5.1.2 State space defined using sectors of front field of vision of the car

The main idea behind this state-space definition is that the front field of vision is divided into a given number of sectors of a given angle. In the code, this is not hard-coded, but a

constant *NUMBER_OF_SECTORS* (by default six so the front field of vision is divided by 30°).

Then, the state can get values 0,1,2 0 is a default, it means that there is no pedestrian in the sector or distance of the pedestrian from the car is greater than some upper threshold. 1 means there is a pedestrian in the sector and its distance from the car is in the interval given by upper and lower threshold and 2 means, that the pedestrian is closer than the lower threshold and is considered near to the car.

The state space is given by all possible combinations of numbers 0,1,2 of a given length. The length is the same as the number of sectors the field of vision is divided into. And a number of state vectors of this state space is then calculated according to equation 2.

$$3^n = 3^6 = 729 \quad (2)$$

Where n is the number of sectors. Initialized Q-Table is very similar to that shown in Tab. 1 The only difference is a bigger number of rows (Tab. 1 consists of 128 rows, Q-table defined using this state space representation is made of 729 rows).

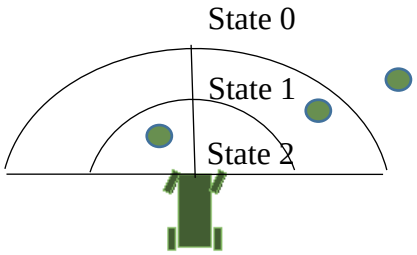

Fig. 20: Car with marked sectors and pedestrians (angle of every sector is the same, in this case, 90°)

To make the decision-making process more clear a simple Q-Table with smaller state space with only two sectors by 90° was generated, the table can be seen in Tab. 2. The n in equation (2) is $n = 2$ so the state space consists only of 9 state vectors (instead of 729.

States of pedestrians in the front field of vision			**Possible actions**			
N	s0	s1	a0	a1	a2	a3
	Left sector	Right sector	Go Forward	Go Backward	Turn Left	Turn right
0	0	0	-1,88359	-3,3311	-4,45218	-2,42678
1	0	1	-4,76737	-0,14146	-2,02942	-2,7696
2	0	2	-2,16294	-2,00945	-1,96646	-0,33793
3	1	0	-3,68806	-2,7473	-3,56019	-3,28458
4	1	1	-3,61843	-2,12605	-1,50367	-2,20062
5	1	2	-0,58493	-4,84069	-2,43092	-2,07347
6	2	0	-0,89335	-0,54965	-1,65248	-4,39696
7	**2**	**1**	**-0,35795**	**-4,73227**	**-0,01736**	**-0,19491**
8	2	2	-2,10883	-4,88364	-3,39145	-0,28639

Tab. 2: Simple Q-Table when the front field of vision is divided into only two sectors

Fig. 20 represents a possible state in which the environment can be, as seen from the picture there are three pedestrians and the state vector is (2,1) – in each sector only the nearest one is taken into account. In Tab. 2 the corresponding row of the Q-table is marked by bold letters. The decision-making algorithm would then pick the action with maximal Q-value. In this case, the max is -0,01736 in a2 column (marked by bold borders of a cell), this means that in the next step the car would pick action 2 and turn left. Whether this is an appropriate action will be decided later and it will be based on a reward the agent gets for such an action.

5.1.3 Functionality testing of the latter state space

For testing purposes, a smaller state space consisting of only two sectors by 90 degrees was generated. In this case, it is visually easy to check whether the state-vector is chosen correctly.

In the simulation loop, a several *print()* commands were put, in each simulation step a current state vector is printed also with a row of the Q-Table which is corresponding to a given state vector and also a line with a statement which action is picked based on the Q-Table row.

The simulation was paused to check whether printed statements are correct, a screenshot of the rendered environment can be seen in Fig. 21.

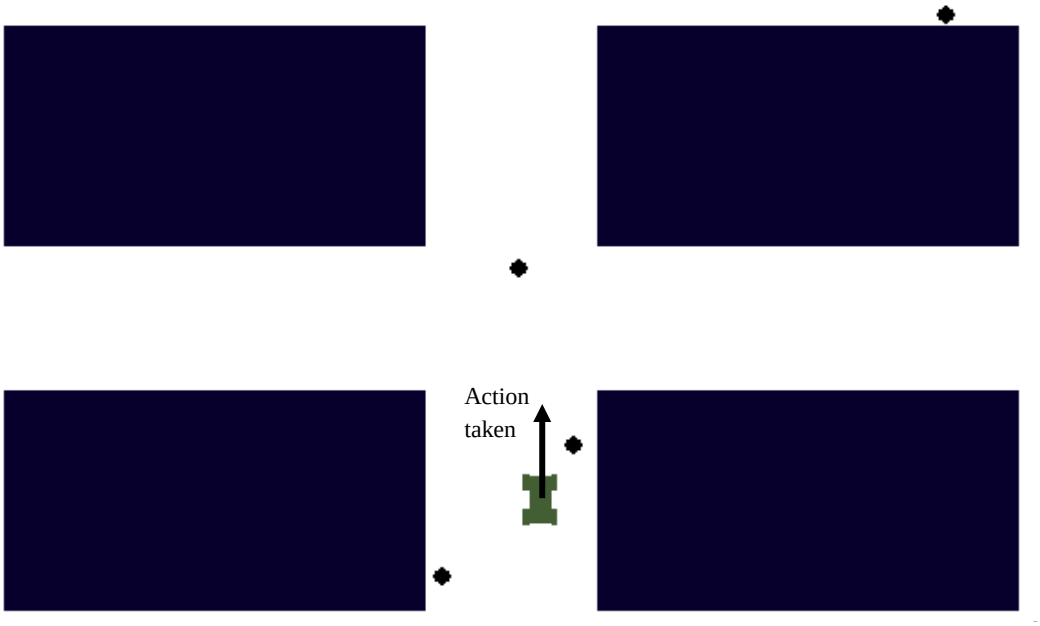

Fig. 21: A screenshot of an environment during simulation – first example

In this simulation step these statements were printed to the console (each row meaning is described below the code snippet):

```
1. Current states are: [0,2]
2. Q-Table row corresponding to current states has index 2, these
   values are: [-1.60328374 -4.8936842 -376143425 -2,46951155]
3. Action with maximal Q-Value is chosen: 0
```

- Row 1 means that there is one pedestrian in sector 1 (indexed from 0) and he is close to the car, it is correct according to Fig. 21.
- Row 2 states that corresponding row has index 2, according to Tab. 2 this is correct as well (Q-values can be and are different because the Q-Table is initialized with random values whenever the simulation is started and also the values might be - and it is almost sure that they were – altered in the process of training
- Row 3 states which action was chosen, it must be the one with the maximal Q-value. As stated in row 2 the maximal value is in the first column so the car picks action with index zero 0 – go forward (as shown in Tab. 2, where all possible actions are indexed from 0 to 3). This is also correct. The action taken is shown in Fig. 21 via the black arrow.

The console output was generated using these three lines of code:

```
1. print(f"Current states are: {current_states}")
2. print(f"Q-
   Table row corresponding to current states has index {q_row}, the
   se values are: {q_table[q_row]}")
3. print(f"Action with maximal Q-Value is chosen: {action}")
```

The code uses Python functionality called *f-String*, or *formatted string literals* which is a way of formatting strings introduced in Python 3.6 [45].

According to the code snippet, the state space is generated correctly, creating state vectors is functional and the decision-making algorithm was set also correctly.

To demonstrate that the algorithm works correctly another environment screenshot was taken (Fig. 22) and corresponding console output is shown.

In order to make the simulation output different from the previous example the whole Q-Table is shown as well.

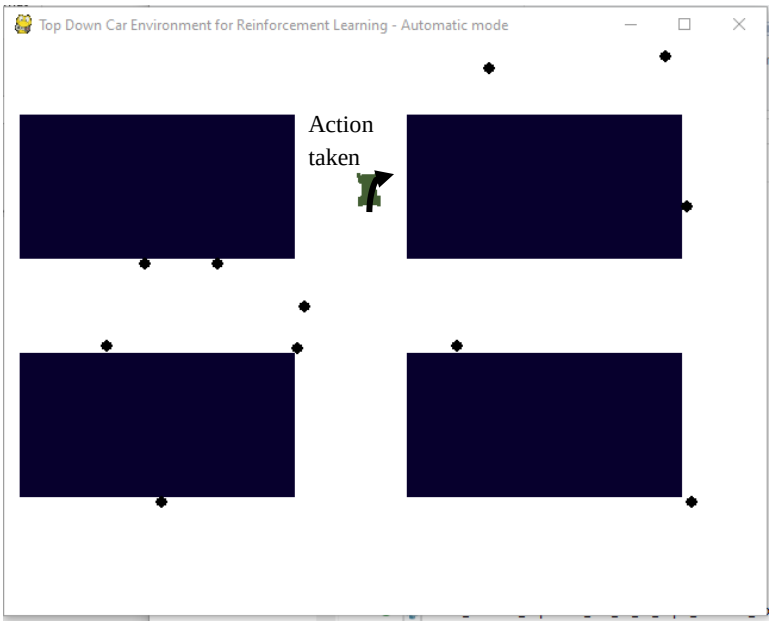

Fig. 22: A screenshot of environment during simulation – second example

```
1. Q-Table is: [array([-0.60568538, -0.16349084, -0.03944161, -
   0.02031068]), array([-3.20479921, -1.82667837, -0.61544554, -
   2.55852629]), array([-2.40663516, -0.68954075, -0.52919228, -
   4.56312428]), array([-4.20554902, -1.69455541, -1.19470289, -
   0.85948568]), array([-3.62712368, -3.25180916, -1.21409192, -
   3.13854511]), array([-0.61549493, -1.97658619, -0.27393306, -
   1.61406332]), array([-1.64311445, -2.33658293, -3.8200347 , -
   3.57131079]), array([-4.02235241, -4.21725415, -2.09287867, -
   0.77741138]), array([-1.30490039, -2.84036692, -1.55760589, -
   4.51158924])]
2. Current states are: [0, 0]
3. Q-Table row corresponding to current states has index 0, these
   values are: [-0.60568538 -0.16349084 -0.03944161 -0.02031068]
4. Action with maximal Q-Value is chosen: 3
```

Code used for generating this output is almost the same as with the previous example, the only difference is that this line was added:

```
1. print(f"Q-Table is: {Q-Table}")
```

Meaning of rows 2, 3, 4 is the same as the meaning of rows 1, 2, 3 in the previous example. Row 1 contains the current values of Q-Table. The meaning of the table is identical to Tab. 2. The only difference is that Tab. 2. Explicitly contains the state space representation. This Q-Table is created based on the state space representation (for example third row - index 2 – is representing values for state vector [0, 2], the state vector is just not printed out.

What concerns row 2, 3, 4, according to Fig. 22. Row 2 is correct (there is no pedestrian near the vehicle), Row 3 is correct as well (states [0, 0] represent the first row of the Q-Table) and finally, row 4 is also correct. The maximal Q-value in this row has index 3, what according to Tab. 2 means turn right (in Fig. 22 shown as a black arrow).

6 CONCLUSION

The goal of this thesis was to create own environment for simulation of an autonomous vehicle and to make the simulator available for the application of artificial intelligence algorithms. Formally the thesis can be divided into two parts, first contains research of currently available tools for autonomous car simulations and of basics of computer physics via physical engines. The second part consists of the development of an environment suitable for testing and development of artificial intelligence algorithms using open-source physical engine Box2D.

The developed environment consists of three types of objects, static bodies representing buildings in a simple city grid, dynamic bodies representing pedestrians that can move around the buildings, and on given locations move from one building to the other, just like real-world people. And the last type of object is a dynamic body representing a car.

This environment was made with an emphasis to use especially for reinforcement learning (or Q-learning to be precise). This basically means that the environment must fulfil several requirements. For example, there must be a way how the environment can be reset when a learning episode is terminated (all of these requirements are explained more in-depth in chapter 4.7.1).

The functionality of the environment was tested and checked using manual mode, in which the car moved in accordance with pressed WASD keys and using automatic mode. This means that a simple algorithm for Q-learning was implemented and ran on this simulator. Both of these ways of testing proved that the environment is developed in a way which allows the simulator to be used for artificial intelligence algorithms testing.

The functionality of the simulator with respect to Q-learning was tested as well and the testing results are satisfying (the testing process is described more in detail in chapter 5.1.3)

6.1 Suggestions for future development

The future development or future improvements might focus especially on two factors

6.1.1 Improving city grid

As seen in Fig. 19 the grid is made of quite simple rectangles, in the future it might be changed for either more complex shapes such as polygons or use some real-world maps, which can be obtained for example from OpenStreetMap project [46].

The former approach is simpler and less realistic, the latter is more realistic but much more difficult for various reasons, for instance, the city grids obtained from OpenStreetMaps are made from complex polygons and Box2D allows to create polygons with a limited amount of edges.

Improving the city grid also requires to improve the algorithm of pedestrian movement. In this environment, the pedestrian can only move left/right by 90 degrees. In order to make them walk around more complex shapes means that turning only by mentioned 90 degrees is not enough.

Also, creating more complex systems of road surfaces can be taken into consideration the vehicle behaves differently when driving on asphalt or through a puddle or gravel roads, etc. This can be done for example by defining polygons with different friction and control whether the car is in contact with these areas.

6.1.2 Improve the driving characteristics of the vehicle

This might, for instance, be focused on turn angle of wheels, in real cars the left and right wheel is turning by a different angle, it uses for example Ackermann steering geometry. Or in modern and more expensive cars both front and rear axles are turning [47].

Also, moving of the car can be made more realistic e.g. by implementing more gradual acceleration or deceleration.

BIBLIOGRAPHY

[1] GKOUMAS, Konstantinos a Anastasios TSAKALIDIS. A framework for the taxonomy and assessment of new and emerging transport technologies and trends. *Transport* [online]. Vilnius: Vilnius Gediminas Technical University, 2019, **34**(4), 455-466 [cit. 2020-01-21]. DOI: 10.3846/transport.2019.9318. ISSN 16484142. Dostupné z: http://search.proquest.com/docview/2314604068/

[2] Tesla Motors Skips Beta Test Phase of Its New Vehicle: Rear-Axle steering. *BetaBreakers.com* [online]. 2017 [cit. 2020-02-02]. Dostupné z: https://www.betabreakers.com/tesla-motors-skips-beta-test-phase-new-vehicle/

[3] Journey. *Waymo.com* [online]. 2020 [cit. 2020-05-21]. Dostupné z: https://waymo.com/journey/

[4] Autonomous 2020. *Ford.com: Corporate* [online]. [cit. 2020-02-25]. Dostupné z: https://corporate.ford.com/articles/products/autonomous-2021.html

[5] Self-Driving Car Technology: We believe in the power of technology. *Uber.com* [online]. [cit. 2020-04-02]. Dostupné z: https://www.uber.com/us/en/atg/technology/

[6] Level 5: Revolutionizing cars, reshaping the future. *Lyft.com* [online]. [cit. 2020-04-02]. Dostupné z: https://level5.lyft.com/

[7] Wikipedia contributors. Death of Elaine Herzberg [Internet]. *Wikipedia, The Free Encyclopedia*; 2020 May 25, 12:40 UTC [cit. 2020-06-01]. Available from: https://en.wikipedia.org/w/index.php?title=Death_of_Elaine_Herzberg&oldid=958733740.

[8] MIT License. *Opensource.org* [online]. [cit. 2020-04-23]. Dostupné z: https://opensource.org/licenses/MIT

[9] SHAH, Shital, Debadeepta DEY, Chris LOVETT a Ashish KAPOOR. *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles* [online]. 2017 [cit. 2020-01-13].

[10] Microsoft AirSim now available on Unity. *Microsoft.com* [online]. [cit. 2020-04-20]. Dostupné z: https://www.microsoft.com/en-us/research/blog/microsoft-airsim-now-available-on-unity

[11] Become a Self-Driving Car Engineer. *Udacity.com* [online]. [cit. 2020-05-02]. Dostupné z: https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013

[12] Virtual Test Drive 2020. *MSCSoftware.com* [online]. [cit. 2020-05-20]. Dostupné z: https://www.mscsoftware.com/Virtual-Test-Drive-2020

[13] CarMaker. *Ipg-automotive.com* [online]. [cit. 2020-05-20]. Dostupné z: https://ipg-automotive.com/products-services/simulation-software/carmaker/

[14] *Sensor Fusion and Tracking for Autonomous Systems* [online]. 2019 [cit. 2020-04-21]. Dostupné z: https://www.mathworks.com/content/dam/mathworks/white-paper/gated/sensor-fusion-and-tracking-for-autonomous-systems-white-paper.pdf

[15] Automated Driving Toolbox. *MathWorks.com* [online]. 2020 [cit. 2020-04-21]. Dostupné z: https://www.mathworks.com/products/automated-driving.html

[16] RoadRunner: VectorZero. *VectorZero* [online]. 2020 [cit. 2020-03-19]. Dostupné z: vectorzero.io

[17] Visualize Automated Parking Valet Using 3D Simulation. *MathWorks.com* [online]. [cit. 2020-06-02]. Dostupné z: https://www.mathworks.com/help/driving/examples/visualize-automated-parking-valet-using-3d-simulation.html

[18] RoadRunner. *MathWorks.com* [online]. 2020 [cit. 2020-04-21]. Dostupné z: https://www.mathworks.com/products/roadrunner.html

[19] Opal-RT: Real-Time simulation | Real-Time solution. *Opal-rt.com* [online]. [cit. 2020-03-05]. Dostupné z: https://www.opal-rt.com/

[20] OpenDRIVE: Managing the road ahead. *OpenDRIVE.org* [online]. [cit. 2020-04-18]. Dostupné z: http://www.opendrive.org/

[21] DOSOVITSKIY, Alexey, Ros GERMAN, Felipe CODEVILLA, Antonio LOPEZ a Vladlen KOLTUN. CARLA: An Open Urban Driving Simulator. *ArXiv.org* [online]. Ithaca: Cornell University Library, arXiv.org, 2017 [cit. 2020-01-12]. Dostupné z: http://search.proquest.com/docview/2076975186/

[22] Coming soon to CARLA Users. *VectorZero* [online]. 2020 [cit. 2020-04-22]. Dostupné z: https://www.vectorzero.io/post/coming-soon-to-carla-users

[23] How to create a new map. *CARLA.org: Read the docs* [online]. 2020 [cit. 2020-04-23]. Dostupné z: https://carla.readthedocs.io/en/0.9.7/how_to_make_a_new_map/

[24] Chipmunk physics. *Chipmunk-physics.net* [online]. 2020 [cit. 2020-04-23]. Dostupné z: https://chipmunk-physics.net/

[25] *Gilbert–Johnson–Keerthi distance algorithm* [online]. [cit. 2020-03-07]. Dostupné z: https://en.wikipedia.org/wiki/Gilbert%E2%80%93Johnson%E2%80%93Keerthi_distance_algorithm

[26] Spatial Indexing. *Chipmunk-physics.net* [online]. 2020 [cit. 2020-04-24]. Dostupné z: http://chipmunk-physics.net/release/ChipmunkLatest-Docs/#CollisionDetection-SpatialIndexing

[27] Wikimedia Commons contributors. File:Bullet Physics Logo.svg [Internet]. Wikimedia Commons, the free media repository; 2020 Apr 20, 15:11 UTC [cited 2020 May 25]. Available

from: https://commons.wikimedia.org/w/index.php?title=File:Bullet_Physics_Logo.svg&oldid=413371990.

[28] JAVORKA Marián: Fyzikální simulace v grafické scéně, diplomová práce, Brno, FIT VUT v Brně, 2011

[29] Angry Birds Classic. *Google Play* [online]. [cit. 2020-03-09]. Dostupné z: https://play.google.com/store/apps/details?id=com.rovio.angrybirds&hl=cs

[30] Erin Catto. *GitHub.com* [online]. 2020 [cit. 2020-04-25]. Dostupné z: https://github.com/erincatto

[31] JORDAN, M I, T M MITCHELL a M I JORDAN. Machine learning: Trends, perspectives, and prospects. *Science* (New York, N.Y.) [online]. 2015, 349(6245), 255-260 [cit. 2020-02-04]. DOI: 10.1126/science.aaa8415. ISSN 00368075. Dostupné z: http://search.proquest.com/docview/1697220242/

[32] STEINER, Margaret C. "Solving" Cancer: The Use of Artificial Neural Networks in Cancer Diagnosis and Treatment. *Journal of Young Investigators* [online]. 2017, December 2017, 2017 [cit. 2020-02-04]. DOI: 10.22186/jyi.33.6.122-124. Dostupné z: https://www.jyi.org/2017-december/2017/11/30/solving-cancer-the-use-of-artificial-neural-networks-in-cancer-diagnosis-and-treatment?rq=Cancer%20Diagnosis%20and%20Treatment

[33] KORYTÁR, L. *Realizace lokalizačního systému pro mobilní robot B2*. Brno: Vysoké učení technické v Brně, Fakulta strojního in□enýrství, 2018. 77 s. Vedoucí doc. Ing. Ji□í Krejsa,Ph.D.

[34] HOFFMANN, D. *Navigace mobilního robotu B2* ve venkovním prostředí. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2019. 67 s. Vedoucí diplomové práce doc. Ing. Jiří Krejsa, Ph.D.

[35] VIOLA, Paul a Michael JONES. Robust Real-Time Face Detection. *International Journal of Computer Vision* [online]. Boston: Kluwer Academic Publishers, 2004, 57(2), 137-154 [cit. 2020-01-12]. DOI: 10.1023/B:VISI.0000013087.49260.fb. ISSN 0920-5691.

[36] BRIDDOCK, David. OpenAI. *Micro Mart* [online]. London: Dennis Publishing, 2016, (1405), 64-66 [cit. 2020-01-13]. ISSN 14730251. Dostupné z: http://search.proquest.com/docview/1786768356/

[37] PLAPPERT, Matthias, Marcin ANDRYCHOWICZ, Alex RAY, et al. *Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research* [online]. 2018 [cit. 2020-01-13].

[38] Wikimedia Commons contributors. File:OpenAI Logo.svg [Internet]. Wikimedia Commons, the free media repository; 2020 Mar 12, 17:46 UTC [cited 2020 May 25]. Available from: https://commons.wikimedia.org/w/index.php?title=File:OpenAI_Logo.svg&oldid=403673312.

[39] WIGGERS, Kyle. OpenAI Five defeats professional Dota 2 team, twice. In: *VentureBeat.com* [online]. April 13th 2019 [cit. 2020-04-05]. Dostupné z:

https://venturebeat.com/2019/04/13/openai-five-defeats-a-team-of-professional-dota-2-players/

[40] Wikimedia Commons contributors. File:Reinforcement learning diagram.svg [Internet]. Wikimedia Commons, the free media repository; 2018 May 24, 00:30 UTC [cited 2020 May 25]. Available from: https://commons.wikimedia.org/w/index.php?title=File:Reinforcement_learning_diagram.svg&oldid=302766739.

[41] F. Codevilla, M. Muller, A. Dosovitskiy, A. L ¨ opez, and V. Koltun. End-to-end driving via conditional ´ imitation learning. arXiv:1710.02410, 2017.

[42] Wikimedia Commons contributors. File:Q-Learning Matrix Initialized and After Training.png [Internet]. Wikimedia Commons, the free media repository; 2018 Nov 12, 01:58 UTC [cited 2020 May 25]. Available from: https://commons.wikimedia.org/w/index.php?title=File:Q-Learning_Matrix_Initialized_and_After_Training.png&oldid=327414267.

[43] TopDown projection of lateral velocity. IForce2D.net [online]. [cit. 2020-02-08]. Dostupné z: https://www.iforce2d.net/image/topdown-projectlateral.png

[44] Wikipedia contributors. Recursion (computer science) [Internet]. Wikipedia, The Free Encyclopedia; 2020 Jun 3, 16:25 UTC [cited 2020 Jun 5]. Available from: https://en.wikipedia.org/w/index.php?title=Recursion_(computer_science)&oldid=960558346.

[45] Lexical analysis: Formatted string literals. *Docs.python.org* [online]. [cit. 2020-06-15]. Dostupné z: https://docs.python.org/3/reference/lexical_analysis.html#f-strings

[46] Wikipedia contributors. OpenStreetMap [Internet]. Wikipedia, The Free Encyclopedia; 2020 May 16, 22:41 UTC [cited 2020 May 25]. Available from: https://en.wikipedia.org/w/index.php?title=OpenStreetMap&oldid=957083704.

[47] Porsche Panamera 4: Rear-Axle steering. Porsche.com [online]. 2020 [cit. 2020-04-25]. Dostupné z: https://www.porsche.com/international/_iceland_/models/panamera/panamera-models/panamera-4/drive-chassis/rear-axle-steering/

LIST OF ABBREVIATIONS

Sorted in alphabetical order

Abbr.	Meaning
AA	Autonomous Automobile
ADAS	Advanced Driver Assistance Systems
ADT	Automated Driving Toolbox
AI	Artifficial intelligence
AirSim	Aerial Informatics and Robotics Simulation
API	Application Programming Interface
AV	Autonomous Vehicle
CARLA	Car Learning to Act
EPA	Expanding Polytope algorithm
FEM	Finite Element Method
GJK	Gilbert-Johnson-Keerthi
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HIL	Hardware in the Loop
HPC	High-Performance Computing
MIT	Massachusetts Institute of Technology
ML	Machine Learning
NPC	Non-Player Character
OOP	Object Oriented Programming
OSM	OpenStreetMap
PC	Personal Computer
PID	Proportional-Integral-Derivative
PPM	Pixels Per Meter
RCP	Remote Procedure Call
RSD	Road Scenario Designer
SAE	Society of Automobile Engineers
TCP	Transmission Control Protocol
USD	United States Dollar
VIL	Vehicle in the Loop
VTD	Virtual Test Drive
VW	Volkswagen

LIST OF FIGURES AND GRAPHS

LIST OF TABLES

APPENDIX

This thesis comes with *.zip* archive which contains following files:

- Python source codes
- Screen video record with running of the python code